University of Alberta

Selective Depth-First Game-Tree Search

by

Yngvi Björnsson ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68547-0

Canada

University of Alberta

Library Release Form

Name of Author: Yngvi Björnsson

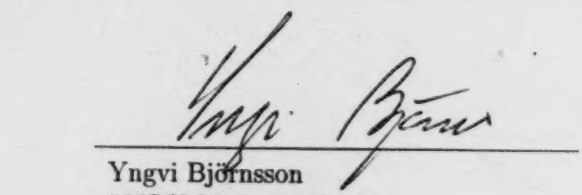
Title of Thesis: Selective Depth-First Game-Tree Search

Degree: Doctor of Philosophy

Year this Degree Granted: 2002

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Yngvi Björnsson
538RH Michener Park
Edmonton, Alberta
CANADA T6H 4M5

Date: April 16th, 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Selective Depth-First Game-Tree Search** submitted by Yngvi Björnsson in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

T.A. Marsland

Dr. T.A. (Tony) Marsland
Supervisor

D F Beal

Dr. Donald F. Beal
External Examiner

Renée Elio

Dr. Renee Elio

Jonathan Schaeffer

Dr. Jonathan Schaeffer

Dr. Gordon Swaters

Date: April 15th, 2002

Til foreldra minna.

Abstract

This thesis continues an ongoing study of algorithms for *selective depth-first expansion* of game trees (for two-person zero-sum perfect-information board games). The question we are primarily concerned with is: how should game-playing programs spend their search effort to maximize the quality of their move decisions?

This is a challenging problem. Early attempts at selective expansion of game trees were not particularly successful, and were replaced by brute-force full-width searches once technological advances in both hardware and software made such approaches feasible. However, such brute-force methods are not sufficient on their own to produce world-class game-playing programs. Therefore, many new algorithms have been proposed for exploring game trees more selectively. On the one hand, there are algorithms that traverse the game trees in a best-first fashion but, unfortunately, these algorithms are neither time nor space efficient and have thus not found a wide use in practice. On the other hand, selective depth-first based search algorithms show more promise.

This work introduces several algorithmic enhancements to state-of-the-art depth-first game-tree search. First of all, we show how speculative pruning can, if applied in a controlled manner, improve the search efficiency of existing game-tree search algorithms — without compromising the returned minimax value. Secondly, a novel domain-independent method for speculative pruning of game trees is presented, and its promise demonstrated in two different search domains. The method has been adapted by some of the world's leading chess-playing programs with good success. Finally, we introduce a method for automatically learning search-control parameters in two-person games, either from online play or by offline analyzes of game positions.

Acknowledgements

I owe my thanks to many people. Without their support and encouragement this endeavor would not have been possible.

First and foremost, I want to thank my supervisor Tony Marsland for his guidance, patience, and support over the years. I have been truly privileged to tap into his wealth of knowledge and experience. I also like to thank the other members of my supervisory committee: Renee Elio, Jonathan Schaeffer and Gordon Swaters. Their invaluable suggestions greatly improved the quality of this work. In particular, I am indebted to Jonathan for his numerous suggestions, constructive criticism, and support over the years. I am also grateful to my external examiner Don Beal. His careful reading and insightful comments further improved the thesis.

The GAMES group at University of Alberta provided a truly unparalleled environment for conducting research into games. Thank you to both past and present members of the group that I have interacted with over the years. In particular, Andreas Junghanns for the years we spent together working on THE TURK and other game-playing programs; Darse Billings for open exchange of ideas while working on the game *Lines of Action*; and Martin Müller for proof-reading a part of the thesis and for many enjoyable afternoon discussions over a cup of coffee.

Finally, big thanks to my parents who have always been there for me. Last and not least, very special thanks to my beautiful wife Gudrun and my admirable son Daniel. Your love and support means more to me than words can say. Thank you from the bottom of my heart!

Table of Contents

1 Introduction **1**

1.1 Game Playing and Search 2

1.2 Contributions 5

1.3 Organization 6

1.4 Publications 7

2 Game-Tree Search **9**

2.1 The Game Tree and Minimax 9

2.2 A Critical Tree and the $\alpha\beta$ Algorithm 13

2.2.1 A Critical Tree 14

2.2.2 The $\alpha\beta$ Algorithm 14

2.3 Algorithmic Enhancements 16

2.3.1 Transposition Table 17

2.3.2 Iterative Deepening 18

2.3.3 Move Ordering 19

2.3.4 Aspiration Windows 19

2.3.5 Minimal-Window Variants 21

2.3.6 Quiescence Search 26

2.4 Best-First Search 27

3 Selective Depth-First Search **29**

3.1 Speculative Pruning 29

3.1.1 Risk Assessment 30

3.1.2 Applicability 35

3.1.3 Cost Effectiveness 35

3.1.4 Domain Dependency 36
3.2 Search Extensions 37
3.3 Related Work 37
3.3.1 The Null-Move Heuristic 37
3.3.2 ProbCut and Multi-ProbCut 38
3.3.3 Singular Extensions 39
3.3.4 Other Methods 39
3.4 Conclusions 40

4 Uncertainty Cutoffs **41**
4.1 Searching a Critical Tree 42
4.2 Uncertainty Cutoffs - Idea 43
4.3 Uncertainty Cutoffs - Algorithm 47
4.4 Experimental Results 51
4.5 Conclusions 54

5 Multi-Cut $\alpha\beta$ Pruning **55**
5.1 Multi-Cut Idea 55
5.2 Multi-Cut Implementation 57
5.3 Multi-Cut Parameters 60
5.4 Experimental Results 61
5.4.1 Criteria Selection 62
5.4.2 Multi-Cut Parameters 65
5.4.3 Multi-Cut in Practice 68
5.5 Related Work 73
5.6 Conclusions 74

6 Learning Search Control **77**
6.1 Introduction 77
6.2 Search Control 79
6.2.1 Search Extensions 79
6.2.2 A Unified View 80
6.2.3 Fractional-Ply Extensions 82

6.3 The Learning System 82
6.3.1 Training Experience 83
6.3.2 Target Function 85
6.3.3 Learning Algorithm 86
6.4 Modeling the Search 89
6.4.1 Cost Model 90
6.4.2 Approximating $B(p, \vec{w})$ and its Partial Derivatives . . . 91
6.5 Experimental Results 94
6.5.1 Test Suite 95
6.5.2 Game Playing 97
6.6 Conclusions 100

7 Learning Search Control Offline **102**
7.1 Introduction 102
7.2 Offline vs. Online Learning 103
7.2.1 How to Estimate the Gradient 104
7.2.2 How Deep to Search? 104
7.2.3 Pros and Cons 105
7.3 Architecture 106
7.3.1 Game-Playing Program 107
7.3.2 Learning Module 108
7.4 Experimental Results 110
7.5 Conclusions 112

8 Concluding Remarks **114**
8.1 Conclusions 114
8.2 Future Work 116

Bibliography **118**

A Game-Tree Viewer **124**
A.1 Viewer 124
A.2 Library 126

B Estimating $B(\vec{w})$ - Example 129

C Gradient of Cost Model 132

D Test Suites 133

D.1 Plaat Test Positions 134

D.2 Bratko-Kopec Test Suite 135

D.3 1001BWC Test Suite 136

D.4 ECM Test Suite 136

D.5 Opening Position Test Suite 137

D.6 Dailey Opening Test Suite 139

List of Figures

2.1 2-ply-deep game tree. 13

3.1 Different risk assessment of subtrees. 32
3.2 Different risk assessment of error propagation. 34

4.1 Critical tree. 43
4.2 Search overhead. 45
4.3 Example position and a corresponding search tree. 46
4.4 Efficiency comparison using the Plaat test positions. 52
4.5 Efficiency comparison using the Brato-Kopec test suite. . . . 54

5.1 Applying the *mc*-prune method at node N. 57
5.2 Search efficiency when $r = 2$. 66
5.3 Decision quality when $r = 2$. 67

6.1 Search-extension schemes - a unified view. 81
6.2 Identifying mistakes. 84
6.3 Approximating $B(p, \vec{w}_1 + \vec{\Delta}_1)$. 93
6.4 Learning results. 95

7.1 The architecture of the offline learning system. 106
7.2 Comparison of online (upper) vs. offline (lower) learner. . . . 111

A.1 Screenshot of the game-tree viewer. 126

B.1 Depth of nodes in a game tree. 130
B.2 Multiple depths of nodes in a game tree. 130

List of Tables

4.1 Uncertainty cutoff results on the Plaat test positions. 53

5.1 Comparison of different schemes for identifying False-cut-nodes. 64
5.2 Comparison of selected schemes using filtered data. 65
5.3 $T_{mc(c,e,r)}$ searches showing the performance of different multi-cut parameter settings relative to a standard search, in both terms of % of nodes searched (Nod%) and problems solved (Sol%). . 68
5.4 80 game multi-cut chess match results. 69
5.5 622 game LOA match result. 71
5.6 311 mini-match LOA result. 73

6.1 Learned weights. 98
6.2 Match results. 100

List of Algorithms

1 $MM(P,d)$ 11
2 $\alpha\beta(P,d,\alpha,\beta)$ 16
3 $ID-asp-\alpha\beta(P,maxdepth,margin)$ 20
4 $\alpha\beta(P,d,\alpha,\beta)$ 22
5 $PVS(P,d,\alpha,\beta)$ 23
6 $MWS(P,d,\beta)$ 24
7 $NS(P,d,\alpha,\beta)$ 25
8 $MTD(P,depth,f)$ 26
9 $ucPVS(P,d,\alpha,\beta)$ 48
10 $ucMWS(P,d,\beta,cut)$ 49
11 $mcMWS(P,d,\beta,cut)$ 59
12 LSC 88
13 LSC-offline 109
14 $ID-PVS(P,d,maxdepth)$ 128

Chapter 1

Introduction

Life is a game, and the only game is real life.
- Anonymous

Ever since the dawn of civilization humans have been fascinated with games; first and foremost for the entertainment value, but also as a mechanism for abstracting more complex real-world scenarios. For example, some of the ancient strategic board games that we know today are believed to have military roots. Instead of having armies go to war, some of the planning and other strategic elements essential for conducting a battle could be practiced on a board.

Today games still serve as a useful mechanism for abstracting real-world situations. In the same way as the chemists and physicists carry out their experiments in closed laboratories where they have full control of their environment, the abstraction power of games provides the computer scientist with an ideal controllable environment for conducting research. For example, game playing is one of the oldest areas of investigation in artificial intelligence (AI) and has been at the forefront of AI research ever since the birth of the first computers, over half a century ago.

Finally, games are interesting on their own. It has proven remarkably challenging task to program computers to play complex board games at the

same level as the best humans. Only in the last decade or so have computers been able to claim a victory over the best humans in non-trivial games of skill such as *checkers* (in 1994 the program CHINOOK became the official checker's World Champion) and *chess* (in 1997 DEEP BLUE defeated the reigning human World Champion in a 6-game exhibition match). However, in other games such as *Go* and *shogi*, humans are still head and shoulders above the best computer players. Much additional research effort is needed before computers can play these games at even a master level.

In recent years, commercial games such as role-playing, real-time strategy, and sport games have become increasingly popular test bed for AI research. These games pose new interesting research problems, such as real-time path finding and behavioral models for non-playing characters. The entertainment business is an important and integral part of today's society, and the computer games industry constitutes a sizable portion of that business — its revenues exceeding those of the film industry [78]. Undoubtedly, for the unseeable future, games will continue to play a prominent role in both AI research and our everyday life.

1.1 Game Playing and Search

Search is fundamental to problem solving. For example, systems for solving planning, scheduling, optimization and constraint-satisfaction problems typically rely on search to a great extend. The same is true for chess playing programs and the like, where the search engine plays a central role in exploring possible move sequences several moves ahead. Unfortunately, the further the programs look a head, the number of game positions they need to analyze grows exponentially.

The prohibitive exponential growth prompted the early game-playing programs to employ highly selective search algorithms. For any given game

position the programs explored only a selected subset of the possible move choices. However, ever increasing advances in computer hardware made full-width searches that examine every possible move choice more and more attractive. By the late 1970s most programs employed such full-width search. On the other hand, when playing chess and other similar games humans are adept at simplifying the search process by reasoning about the choices and then selecting a few prime candidates. Strong players can analyze forced continuations to a great depth; often far deeper than a full-width program can accomplish (even on modern computer hardware). Therefore, over the years various additional enhancements have been introduced to allow full-width searches to be more selective. Interesting continuations are typically explored beyond the normal search depth, while less interesting alternatives are terminated prematurely. In chess, for example, it is common to resolve forced situations, such as checks and recaptures, by searching them more deeply.

Nowadays selective-search strategies are an essential part of most game-playing programs. This is, in part, a consequence of the fact that further full-width search (made possible by increased hardware speed) exhibits diminishing returns in terms of an increased playing strength [40]. Instead, more selective approaches show a better promise. For a brief period selective strategies that expand game trees in a best-first fashion spurred a considerable interest among the research community. However, it soon became apparent that the overhead necessary for best-first expansion of game trees more than offsets the possible benefits. Thus, best-first search strategies — while interesting in theory — have not gained a widespread use in practice. Therefore, in the last decade or so the focus has shifted and selective depth-first search strategies have instead become one of the more active research areas in game-tree search [2, 7, 25, 39, 17, 27].

One of the most prominent example of this is the DEEP BLUE chess pro-

gram. The success of the program can in part be explained by the impressive hardware architecture, allowing the program to analyze on average over one-hundred million chess positions per second. However, that on its own is not sufficient. Additional search enhancements were necessary to best take advantage of the raw search speed. On the software side, the DEEP BLUE research team designed innovative selective search algorithms based on extending forced lines of play. The decision to use a highly non-uniform search was in part prompted by the observation that strong chess players were on occasions able to out-search previous versions of the program that employed a more uniform search. In the DEEP BLUE teams words: "Our experiments showed that DEEP BLUE typically sacrificed two ply of full-width search in order to execute the selective search algorithms. The reason that this was deemed acceptable was that DEEP BLUE had sufficient searching power that this loss of two ply still left enough full-width search depth to satisfy our insurance needs." [27]. From this one can deduce that around 95% of all game positions analyzed by the program were in selectively extended lines!

A similar story can be told for most other world-class game-playing programs. For example, LOGISTELLO, the Othello [1] program that dominated the world scene for many years employed an aggressive pruning scheme during its search. Not only did the program win almost every single computer Othello competition it participated in, but one of its many accomplishments was to defeat the human Othello World Champion in a 6-game match in 1997, winning each and every game.

In this thesis we continue the ongoing investigation of selective expansion of game trees. For practical reasons we limit the scope of the thesis to only depth-first search. The fundamental question we are primarily concerned with is: *Given that there is a limited amount of time to make a move decision, how*

[1] Othello is a registered trademark of Tsukuda Original, licensed by Anjar Co.

- *should game-playing programs best spend their search effort to maximize their move decision quality?*

1.2 Contributions

This thesis enhances our understanding of selective depth-first game-tree search and contributes to the state-of-the-art in several ways, including:

- An additional insight into speculative pruning is provided. Whereas existing pruning methods are mainly concerned with the likelihood of an erroneous pruning decision being made in a local subtree, we show that it is equally important to assess the likelihood that an erroneous pruning decision, if made, will propagate up the tree and thus affect the move decision at the root.

- The above observation forms the basis of a new domain-independent pruning method, *Multi-Cut*. We introduce the method here and experiment with it using two different games as a test bed. Our experimental results demonstrate the promise of the method: the playing strength of the programs we tested improves significantly. Additionally, to our knowledge at least two of the top commercial chess programs have subsequently incorporated the method with good results.

- Over the decades new improved variants of depth-first game-tree search algorithms have seen the light of day. These variants show a slight improvement in search efficiency, that is, they explore fewer nodes while reaching the same move decision. We show how speculative pruning — if applied in a controlled manner — can even further improve the search efficiency without affecting the move decision. We call the enhancement *Uncertainty Cutoffs*.

- We introduce a novel method for learning search-extension parameters. We show how the search-control problem can be formulated as a function approximation task, thus allowing us to use standard machine-learning methods for tuning the search-control parameters. The new learning method can be applied during either online play or offline analyzes of game positions. Furthermore, we have implemented and made publicly available an abridged version of the learner as a stand-a-lone application. It has the nice property that almost any search based game-playing program can be "plugged" into the learner as a separate module (requiring only a few trivial modifications to the game-playing program itself).

- Finally, we developed and made public a software tool for visualizing game trees. This tool has proved an invaluable aid to us in understanding and analyzing game trees and, in particular, debugging the search process. Hopefully others will also find this software tool equally helpful.

1.3 Organization

Chapter 2 gives an overview of the most common search techniques used in (search-based) two-person games, emphasizing the methods that have withstood the test of time and are being employed by most contemporary game-playing programs.

In the next few chapters we investigate pruning as a way of adding selectivity to full-width search. Chapter 3 focusses on such enhancements, the intent of the chapter being twofold. First, to give an overview of existing pruning methods and to identify their shortcomings. Second, to identify and summarize properties that are important to consider when pruning game trees, thereby enhancing our understanding of pruning. The algorithms introduced in later chapters were in part prompted by some of the insights gained during

this investigation. In Chapter 4 we present *Uncertainty Cutoffs*, a minor enhancement to standard game-tree search. Chapter 5 introduces *Multi-Cut*, a new selective pruning method.

In addition to pruning, search extensions are commonly used to make full-width search more selective. In the second half of the thesis we take a close look at search extensions. Chapter 6 introduces a novel method for learning control parameters in adversary search, in particular parameters for controlling search extensions. In Chapter 7 a simplified version of the learning algorithm is given. Although being more limited, in the sense that it is restricted to learn during offline analyzes of game positions, it has other desirable properties.

Finally, in Chapter 8 we provide conclusions and discuss some of the outstanding research issues. The appendixes provide additional experimental data and proofs. Furthermore, we present a software tool that we developed for visualizing game trees, and give examples of its use.

1.4 Publications

Chapter 4 is based on a paper "Searching With Uncertainty Cutoffs" that appeared in the International Computer Chess Association Journal [19]. The idea was first presented at the Advances in Computer Chess 8 conference, Maastricht, 1996.

The Multi-Cut idea from Chapter 5 was first presented in 1998 at "The First International Conference on Computers and Games", Tsukuba, Japan [14]. A revised and extended version was published in the Theoretical Computer Sciences journal [17]. An overview article of pruning in game-tree search, including topics from Chapters 3 and 5, appeared in the Information Sciences Journal [15].

Chapter 7 is based on an article "Learning Search Control in Adversary Games" that was presented at a computer games conference in Paderborn,

Germany in 1999. The conference proceedings were recently published as a book "Advances in Computer Games 9" [16].

A paper, "Learning Control of Search Extensions", that is based on Chapter 8 was presented at the "Joint Conference on Information Sciences", Research Triangle Park, North Carolina, March 8-14, 2002 [18].

Chapter 2

Game-Tree Search

> *The passion for playing chess is one of the most unaccountable in the world. It slaps the theory of natural selection in the face. It is the most absorbing of occupations. The least satisfying of desires. A nameless excrescence upon life. It annihilates a man. You have, let us say, a promising politician, a rising artist that you wish to destroy. Dagger or bomb are archaic and unreliable - but teach him, inoculate him with chess.*
>
> *H.G. Wells, 'Certain Personal Matters', 1898*

We are concerned with two-person zero-sum perfect information board games, such as chess, checkers, and Othello (and many others). In this chapter we give an overview of game-tree search for that type of games. We will introduce the basic terminology as well as describing the best established search methods and enhancements. We are primarily focused on the methods that have withstood the test of time and are being employed (almost universally) in contemporary game-playing programs. However, where appropriate we will mention other (less successful) approaches and direct the reader to the relevant literature.

2.1 The Game Tree and Minimax

The *game-theoretic value* (or *game value* for short) of a two-person zero-sum perfect-information game is the outcome when both players play perfectly, and can be found by recursively expanding all possible continuations from the

initial game state, until states with a known outcome are reached (so-called *terminal states*). The ***minimax rule*** is then used to propagate those outcomes back to the initial state [79]. Using the minimax rule, *Max*, the player to move at the root, tries to optimize its gains by always returning the maximum of its children's values. The other player, *Min*, tries to minimize *Max's* gains by always choosing the minimum value (thereby maximizing its own gains). However, for zero-sum games one player's gain is the others loss. Therefore, by evaluating the terminal nodes from the perspective of the player to move and negating the values as they back up the tree, the value at each interior node can be treated as the merit for the player who's turn it is to move in that state. This formulation is referred to as ***negamax*** [44] and has the advantage of being simpler and more uniform, since both players now are handled the same way, that is, both maximize the backed-up values. We use this formulation in all our subsequent discussion. The state space expanded this way is a tree, often referred to as a *game tree*, where the root of the tree is the initial state and terminal states are the leaf nodes.

In theory, at least, the value of a game can be found as described above. However, the exponential growth of game trees expanded this way is prohibitively time expensive. For example, the number of nodes in the game tree from the initial chess position is estimated to be around 10^{43} [74], more than the number of atoms in the universe! Therefore, in practice, game trees are instead expanded only to a limited depth and the resulting leaf nodes are assessed. In that case the true value of the leaf nodes are generally not known, so the assessment is instead an estimate that measures the "goodness" of the state. This estimate is typically a scalar number and the higher the number, the more likely the state is to lead to a win for the player to move in that state. The exact meaning of the estimate is really not that important from the search point of view — the purpose is simply to provide a ranking of the leaf nodes

Algorithm 1 $MM(P, d)$

```
1: if d ≤ 0 or isTerminal(P) then
2:    return evaluate(P)
3: end if
4: best ← −∞
5: M ← generateMoves(P)
6: for all m_i ∈ M do
7:    make(P, m_i)
8:    v ← −MM(P, d − 1)
9:    retract(P, m_i)
10:   if v > best then
11:      best ← v
12:   end if
13: end for
14: return best
```

to guide the search to the most desirable state. However, it is important to understand that the values are estimates and may be in error. Despite that, the estimates are traditionally treated as if they were true values and they are propagated back up the tree in the same manner using the minimax rule. The value backed up to the root this way is generally called the *minimax value* of the game tree.

The *minimax* algorithm (in the *negamax* formulation) is outlined as Algorithm 1. The parameters P and d represent the current game state (game position) and the remaining search depth, respectively. First, the algorithm checks if the predefined search-depth limit is reached or if the current game position is a terminal state (function $isTerminal(P)$). In both cases the value of the current position as assessed by the $evaluate(P)$ function is returned (lines 1–3). The returned value is from the perspective of the side to move in that position, a positive score indicating an advantage and a negative score a disadvantage. Otherwise, the algorithm generates all possible moves (or actions) from the current game position (function $generateMoves(P)$), and then iterates through the moves looking at each in a turn (lines 6-13). Within the loop,

the current game state P is updated by executing the current move (function $make(P, m_i)$), and then the *minimax* function is called recursively with the updated game state and the remaining reduced search depth as arguments. The returned value is recorded in a variable, here called v. Referring back to our discussion of negamax, we notice that the returned value is negated to make it reflect the merit from the perspective of the player to move. The $retract(P, m_i)$ function restores the current game state by undoing the last move. The value v is then compared to the previously known best value (recorded in variable *best*), and if it is better (higher) it becomes the new best value. Finally, the value of *best* is returned, indicating that this is the best value that the player to move can achieve from that position. In the above (and subsequent) discussion the terms game state and game position (or simply state and position) are used interchangeably. Also, the notation introduced here will be used for the remaining algorithms found in this thesis.

Figure 2.1 shows a game tree for a hypothetical game as expanded by the minimax algorithm when called as $MM(A, 2)$. The root position A is searched to the depth of 2-ply (the term *ply* refers to a half-move, that is, a move by one side). Each node in the tree represents a game state and the edges between nodes represent moves leading from one state to the next. Without a loss of generality we assume that there are exactly 3 actions possible in each game state. The minimax algorithm expands the tree recursively in a left-to-right depth-first manner, first expanding move a_1 leading to game position B. There move b_1 is expanded first, leading to position C where the 2-ply depth limit is reached. The merit of position C is now assessed using the evaluation function ($evaluate(C)$). Assume that it returns the value +5, indicating that position C is slightly advantageous for the player that has the move, in our case *Max*. The algorithm now backtracks, returning the value +5 back to the previous level (node B) where it becomes -5 (remember that in the negamax framework

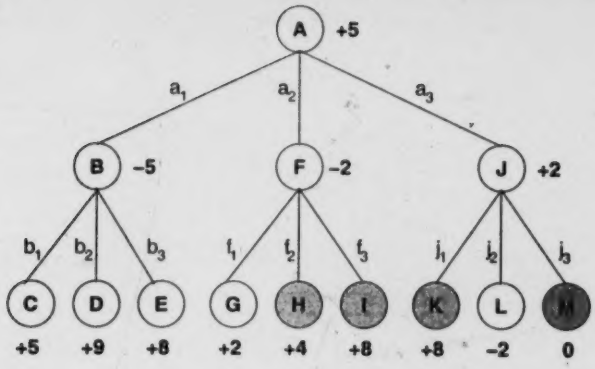


Figure 2.1: 2-ply-deep game tree.

the values are negated as they are backed up so they always indicate the merit from the perspective of the player to move). At node B moves b_2 and b_3 are expanded next in a similar fashion. In our example the moves b_1, b_2, and b_3 get the values -5, -9, and -8, respectively (from *Min's* perspective). The maximum of these values, or -5, is propagated back up to A, where move a_1 gets the value $+5$ (negated again). Next move a_2 is expanded and so forth (the remaining nodes are expanded in the order F, G, H, I, J, K, L and M), finally resulting in the minimax value of $+5$ being determined at the root and a_1 being the move that leads to the maximal score.

2.2 A Critical Tree and the $\alpha\beta$ Algorithm

The minimax algorithm exhaustively explores *all* possible moves for both players when determining the minimax value. However, it is not necessary to investigate all the possible continuations in the game tree to determine its minimax value — only a subtree of the game tree needs to be explored, a so-called *critical tree.*[1]

[1]The term *minimal tree* is also used to refer to this subtree. However, we prefer to use the term *critical tree* because for non-uniform trees a critical subtree is not necessarily minimal in the sense that it contains the fewest nodes.

2.2.1 A Critical Tree

The minimax value of a game tree depends only on the value of nodes in a critical tree; no matter how the remaining nodes are assessed, the minimax value at the root of the tree does not change! To better understand this, let us consider the tree in Figure 2.1 again. After searching the first move *Max* (the player at the root) knows that a value of +5 can be achieved by playing the move a_1. However, it is possible that *Max* could do even better by playing one of the alternative moves. Now *Max* considers the move a_2, *Min* replies with f_1 and the resulting position G is evaluated as +2 in favor of *Max*. Thus, if *Max* were to play move a_2, *Min* could limit *Max's* gains to a mere +2 by replying with move f_1. Consequently *Max* will clearly prefer move a_1 over move a_2, where *Max* is guaranteed a value of +5. There is indeed no need to explore replies to move a_2 any further. Considering the remaining moves at F is simply irrelevant, because move f_1 is sufficient for refuting move a_2. Intuitively, one can say that once a refutation to the opponent's move is found there is no need to look for further refutations! Similar arguments can be used to show that move j_3 at node J can be ignored. These moves and their subtrees, shown shaded in the figure, are thus not a part of a critical tree. It is worth mentioning that there can exist many different critical trees for any game tree. In our example, at node F the move f_2 could equally well have replaced move f_1 to form a different critical tree. We will discuss critical trees in some more detail in a later chapter where it becomes directly relevant to the concepts being introduced.

2.2.2 The $\alpha\beta$ Algorithm

The $\alpha\beta$ algorithm [23, 58] is based on the observation that the minimax value can be found by searching only a part of the game tree, namely an aforementioned critical tree. As noted before, the minimax value of a game tree depends

only on the nodes in a critical tree: no matter how the other nodes are assessed, the minimax value of the tree does not change. The problem is that we do not know beforehand which nodes belong to a critical tree. However, during the search we can establish lower and upper bounds on the range of possible minimax values that subtrees belonging to a critical tree must necessarily have. These bounds are then used to effectively prune the subtrees whose value falls outside the established range, knowing they cannot belong to a critical tree. Specifically, once we have searched at least one child of some node n, we have a lower bound on the actual minimax value of that node. Moreover, if this value exceeds the upper bound already established for subtrees belonging to a critical tree, the remaining children nodes of n need not be searched.

The $\alpha\beta$ algorithm is shown as Algorithm 2. It keeps track of the aforementioned lower and upper bounds via two parameters named α (alpha) and β (beta), respectively. The $\alpha\beta$ routine is called recursively and because we use the negamax formulation, the return value and the bounds are negated in every call. Furthermore, the α and β bounds are switched around, that way the parameter β is always an upper bound for the player to move, so we don't need to distinguish between α and β cutoffs (see below). The aforementioned pruning condition is checked at lines 12–14. If a move returns a value greater or equal to β, the local search terminates at that particular node; this is often referred to as a β-cutoff. To ensure that the minimax value of the tree will be found, the algorithm is initially called with the values of α and β as $-\infty$ and ∞, respectively.

The number of nodes that the $\alpha\beta$ algorithm expands compared to minimax search depends on the order in which the moves are considered. Generally speaking, we want to explore good moves as early as possible; that way tight bounds are established early, thus allowing for more of the remaining search tree to be pruned via β-cutoffs. In the worst case the $\alpha\beta$ algorithm expands the

Algorithm 2 $\alpha\beta(P, d, \alpha, \beta)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: best ← α
5: M ← generateMoves(P)
6: for all m_i ∈ M do
7:   make(P, m_i)
8:   v ← −αβ(P, d − 1, −β, −best)
9:   retract(P, m_i)
10:  if v > best then
11:    best ← v
12:    if best ≥ β then
13:      return best
14:    end if
15:  end if
16: end for
17: return best
```

whole game tree exhaustively just like the minimax algorithm, whereas in the best case only a critical tree is expanded (in which case the number of nodes is approximately the square root of the number of nodes the minimax algorithm visits). Knuth and Moore provide an analysis of the search complexity of the $\alpha\beta$ algorithm [44].

2.3 Algorithmic Enhancements

Over the years, a number of enhancements have been proposed to the basic $\alpha\beta$ algorithm. The first class of enhancements extends the algorithm by improving its search efficiency, mainly by trying to take advantage of a good move ordering. The goal is to make the algorithm behave as closely as possible to its best-case behavior. However, these enhancements do not alter in any way the minimax value returned by the algorithm. The second class of improvements enhances the decision quality of the algorithm by expanding selected continuations more deeply, while discarding other less promising lines. In this section

we focus only on the first class of improvements. Later chapters discuss in detail the second type of improvements.

2.3.1 Transposition Table

In chess and many other games, different move sequences can lead to identical positions. Therefore, the search space is strictly speaking not a tree, but rather a graph. However, depth-first search algorithms typically do not treat the search space as such, but instead use a big table, a so-called *transposition table*, to keep track of possible transpositions to duplicated subtrees. After exploring a game position, the search stores information about it in the table. If that position is encountered again in the search via an alternative move path, it may not be necessary to search it again — its value can be retrieved from the table. The use of transposition tables was first introduced in the MAC-HACK chess program [37]. The table is indexed by hashing game positions, using an efficient hashing function such as the one introduced by Zobrist in 1970 [83].

Because the transposition table can hold only a small fraction of the actual game positions encountered during the search, a replacement scheme is needed to decide which positions to keep in the table. One popular replacement scheme uses a two-level transposition table. This table stores two positions for each hash entry: the most recent position hashed into that entry, and the position that was searched the deepest. Typical information to store for each position in a transposition table is the value of the position, type of the value (i.e. true value, upper bound, or lower bound), the height of the subtree searched from that position, the best move in the position, and the full hash key. Additional information is sometimes kept; for example more sophisticated replacement schemes store an age stamp that records when an entry was inserted into the transposition table. For a detailed discussion of the use of transposition tables in games see Breuker [22].

2.3.2 Iterative Deepening

When using a depth-first search it is necessary to decide beforehand how deeply to search. This makes it difficult to estimate how long the search will take. The original impetus behind using iterative deepening was simply to get a better time control mechanism. By gradually increasing the search depth one can better decide how long the search will take and when to stop the search. Iterative deepening first does a 1-ply search, then a 2-ply search, and so forth until the time allotted for the search is up. The time each successive iteration takes grows exponentially with the search depth, thus the effort spent in the earlier iterations is relatively small compared to the time for the last iteration. The additional search introduced by iterating on the search depth is therefore small. Furthermore, when used in combination with a transposition table, information about the previously seen best moves is kept between iterations. This leads to better move ordering (see next sub-section), most often resulting in the iterative-deepening approach searching fewer nodes in total than the non-iterative approach! The (now legendary) chess program CHESS 4.5 [75] was one of the first programs to use this technique, in the early 1970's. The technique of iterative-deepening search later found its way into other AI domains, such as theorem-proving [76] and single-agent search [46].

Iterative deepening can also be applied at internal nodes in the search tree, a scheme referred to as *internal iterative deepening* [2]. However, programs generally do not apply the technique at every internal node, but rather at selective places. For example, internal iterative deepening is commonly applied on the principal variation (see later) if the best move is not found in the transposition table [2].

2.3.3 Move Ordering

As is well known, good move ordering is of paramount importance for the $\alpha\beta$ algorithm to search efficiently. Therefore, many move-ordering schemes have been developed. One technique, especially useful when used in association with iterative deepening, is to store the best move in the transposition table. When a node is revisited on subsequent iterations, this move is always tried first. The rationale is that a move previously found to be good in a position is also most likely to be good when the position is searched to a greater depth. Another useful heuristic, in chess at least, is to try capture moves before non-capture moves, because often an easy refutation is found by an obvious capture.

The *killer move* [75] and the *history heuristic* [71] are two move-ordering schemes that are also widely used. The former keeps, for each depth level in the tree, a list of moves that have most frequently caused a cutoff. When generating moves in any given position, the killer moves at the current level – if legal for that position – are sorted such that they are early in the move list. The history heuristic keeps global information about moves, indexed by the side-to-move, and the from and to square. Whenever a move causes a cutoff it receives a credit. The closer the move is to the root of the search tree, the more credit it gets. The table keeps track of the accumulated credit of moves, and in any given position moves with a high credit are searched earlier than moves with a low credit. More recently there have been attempts to have programs automatically learn good move-ordering schemes [38, 47].

2.3.4 Aspiration Windows

The observation that the narrower the $\alpha\beta$ window the better the $\alpha\beta$ algorithm performs, because of additional cutoffs, led to the idea of an *aspiration window* search [23, 51]. Given that one can reasonably estimate a range where the minimax value is expected to lie, then instead of calling the $\alpha\beta$ algorithm

Algorithm 3 $ID - asp - \alpha\beta(P, maxdepth, margin)$

```
1: α ← −∞
2: β ← +∞
3: for depth ← 1 to maxdepth do
4:    v ← αβ(P, depth, α, β)
5:    if v ≤ α then
6:       v ← αβ(P, depth, −∞, v)
7:    else if v ≥ β then
8:       v ← αβ(P, depth, v, +∞)
9:    end if
10:   α ← v − margin
11:   β ← v + margin
12: end for
```

with an initial window of $(-\infty, +\infty)$, a narrower window can be used. If the value returned by the search falls inside the estimated window, it is the true minimax value and considerable search effort may be saved. However, there is a possibility that the bound estimates are poor, in which case the search would return a value outside the aspiration window. If that happens, then the returned value, v, is *not* the minimax value, but instead a bound on the minimax value: an upper bound if the search fails low ($v \leq \alpha$), a lower bound if it fails high ($v \geq \beta$). To determine the correct minimax value the search must be repeated, this time with a more appropriate window. However, if we have a good estimate of the minimax value before starting the search, few re-searches are necessary and the savings resulting from having a narrow initial window will outweigh the additional search effort introduced by the occasional re-searches. We can usually get a reasonably accurate estimate on the minimax value, especially when using iterative deepening: simply use a narrow window around the minimax value returned by the previous search iteration, as shown in Algorithm 3. The algorithm demonstrates the use of aspiration window in association with iterative deepening. The root game position (P) is iteratively searched to depth *maxdepth*, using an aspiration window of $\pm margin$ around the value returned from the previous iteration.

A small enhancement to the $\alpha\beta$ algorithm, called *fail-soft* $\alpha\beta$ [33], is beneficial when used in association with aspiration-window search (and minimal-window search as we will see later). When failing to find any good move (i.e. no move has a value larger than α) the basic $\alpha\beta$ algorithm returns the value α. In such cases, the improvement introduced in fail-soft $\alpha\beta$ is to return the value of the best move rather than α. The benefit of this approach is that in case the backed-up value falls outside the initial search window, we have a tighter upper or lower bound on the correct minimax value, depending on whether we failed low or high, respectively. This is reflected in Algorithm 3 where the value v returned by the search is used as a bound for the re-search. The $\alpha\beta$ algorithm with the fail-soft improvement embedded is shown as Algorithm 4 below. Because the value of *best* can now possibly be lower than the value of α, we cannot use it as the upper bound for the recursive $\alpha\beta$ function call as we did before. Instead a new variable *lower* is introduced for that purpose, and it always keeps the maximum value of α and *best*. The fail-soft enhancement has become an integral part of the $\alpha\beta$ algorithm — it costs nothing and can help prune the tree. In our future discussion we assume the $\alpha\beta$ algorithm and its *minimal-window variants* are all fail-soft enhanced.

2.3.5 Minimal-Window Variants

The idea of an aspiration window can be taken even further. When the $\alpha\beta$ algorithm is used with the aforementioned enhancements (e.g. iterative deepening, transposition tables, and a good move-ordering scheme) it first expands the path it believes is the best line of play and that turns out to be the case more often than not. This line, called the *principal variation*, is searched with a wide window, typically $(-\infty,\infty)$ unless aspiration-window search is used. Now, because we really expect this to be the best line of play, all we want is to show that the alternative moves are inferior. To do so it is sufficient to search

Algorithm 4 $\alpha\beta(P, d, \alpha, \beta)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: best ← −∞
5: lower ← α
6: M ← generateMoves(P)
7: for all m_i ∈ M do
8:   make(P, m_i)
9:   v ← −αβ(P, d − 1, −β, −lower)
10:  retract(P, m_i)
11:  if v > best then
12:    best ← v
13:    if best ≥ β then
14:      return best
15:    end if
16:    lower ← max(α, best)
17:  end if
18: end for
19: return best
```

them with a *minimal window* around the score returned by the principal variation. A window where the α and β parameters are set to be "consecutive" values is called a minimal-window (sometimes also referred to as a zero-width-window or a null-window). The window is $(v, v + \epsilon)$ where ϵ is the smallest granularity of the value returned by the evaluation function. For example, if $evaluate(P)$ returns only integer values, ϵ would be set equal to 1. This results in efficient searches (because of the small window there will be additional cutoffs) and only occasionally, when an alternative move really turns out to be better, is a re-search with a wider window necessary. This idea was first proposed in the *Principal Alpha-Beta* algorithm [33], and the *Scout* algorithm [61]. They were later refined and reworked into the $\alpha\beta$ framework, materializing in the algorithms that are today almost universally used for searching game trees: *Principal Variation Search* [50] and *NegaScout* [64].

Algorithms 5 and 6 show the two functions that constitute the *Principal*

Algorithm 5 $PVS(P, d, \alpha, \beta)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: M ← generateMoves(P)
5: make(P, m_1)
6: best ← −PVS(P, d − 1, −β, −α)
7: retract(P, m_1)
8: if best ≥ β then
9:   return best
10: end if
11: lower ← max(α, best)
12: for all m_i ∈ M|i > 1 do
13:   make(P, m_i)
14:   v ← −MWS(P, d − 1, −lower)
15:   if v > lower and v < β then
16:     v ← −PVS(P, d − 1, −β, −v)
17:   end if
18:   retract(P, m_i)
19:   if v > best then
20:     best ← v
21:     if best ≥ β then
22:       return best
23:     end if
24:     lower ← max(α, best)
25:   end if
26: end for
27: return best
```

Variation Search algorithm [52]. PVS, the main driver, explores the expected principal variation, while the MWS part visits all the alternative nodes, using the lower bound established in PVS to reduce its search effort. The algorithm starts by recursing down what it believes to be the principal variation (line 6). Once the depth limit is reached it starts backtracking up again, but now the sibling moves are initially searched with a minimal window around the value returned by the principal move (line 14).[2] If the minimal-window search

[2]More precisely, because we are using the fail-soft enhancement, the alternatives are searched with the value of the current best move or α, whichever is the larger (recall our previous discussion of the use of the *lower* variable).

Algorithm 6 $MWS(P, d, \beta)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: best ← −∞
5: M ← generateMoves(P)
6: for all m_i ∈ M do
7:   make(P, m_i)
8:   v ← −MWS(P, d − 1, −β + ε)
9:   retract(P, m_i)
10:  if v > best then
11:    best ← v
12:    if best ≥ β then
13:      return best
14:    end if
15:  end if
16: end for
17: return best
```

(MWS) fails low that move has been proved inferior. Only occasionally one of the siblings returns a better value (fails high) and in that case the algorithm re-searches that move to establish a new principal variation (lines 15–17). The MWS is basically a simplified $\alpha\beta$ search. Note that there is really no need to pass around the α bound because it can be set to $\beta - \epsilon$.

Apart from *Principal-Variation Search*, the other algorithm of choice for searching game trees is *NegaScout*. The two algorithms are essentially equivalent search wise; they expand an identical tree.[3] The latter is simply a more compact formulation, using one recursive routine instead of two. In later chapters where we introduce our new enhancements, we demonstrate them using the PVS/MWS algorithm, although they could equally well be implemented in the *NegaScout* algorithm. For those who are curious about its formulation

[3]When the *NegaScout* algorithm was originally introduced it had a small refinement added. If a minimal-window search fails high with a remaining search depth of two or less, there is no need to re-search that move with a wider window to establish its correct minimax value. A value returned from such shallow trees, although outside the search window, is necessarily a correct minimax value. However, this refinement is hardly ever used in practice, both because the node savings are negligible and, more seriously, it is not guaranteed to work correctly when used in combination with search extensions.

Algorithm 7 $NS(P, d, \alpha, \beta)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: M ← generateMoves(P)
5: make(P, m_1)
6: best ← −NS(P, d − 1, −β, −α)
7: retract(P, m_1)
8: if best ≥ β then
9:   return best
10: end if
11: lower ← max(α, best)
12: for all m_i ∈ M|i > 1 do
13:   make(P, m_i)
14:   v ← −NS(P, d − 1, −lower − 1, −lower)
15:   if v > lower and v < β then
16:     v ← −NS(P, d − 1, −β, −v)
17:   end if
18:   retract(P, m_i)
19:   if v > best then
20:     best ← v
21:     if best ≥ β then
22:       return best
23:     end if
24:     lower ← max(α, best)
25:   end if
26: end for
27: return best
```

we show the *NegaScout* algorithm as Algorithm 7, but without explanations (basically, it is identical to the PVS function except the MWS function calls are replaced with calls to the function itself).

Finally, the newest addition to the minimal-window $\alpha\beta$-variant family is $MTD(f)$ [63], shown as Algorithm 8. The algorithm is admirably simple and elegant: a driver that repeatedly calls minimal-window searches to gradually narrow the range between the lower and upper bounds. The additional argument f is the bound used for the initial minimal-window search, typically taken from the value returned by the previous iteration. Given that we have a

Algorithm 8 $MTD(P, depth, f)$

```
1: v ← f
2: α ← −∞
3: β ← +∞
4: repeat
5:    if v = α then
6:       bound ← v + ε
7:    else
8:       bound ← v
9:    end if
10:   v ← MWS(P, depth, bound)
11:   if v < bound then
12:      β ← v
13:   else
14:      α ← v
15:   end if
16: until α = β
17: return v
```

reasonably accurate initial guess, only a few minimal-window searches are necessary to zoom in on the correct minimax value. Although the algorithm has proved itself to search slightly more efficiently than its other more widespread minimal-window variants, it has not yet found itself in a wide practical use (there are some practical issues that need to be addressed when implementing the algorithm in a game-playing program, like how to retrieve the principal-variation and how to handle unreliable bounds caused by window dependent search extensions).

2.3.6 Quiescence Search

Having searched an initial game position to the intended maximum depth, some of the positions that arise are volatile and hard to statically evaluate. For example in chess, if the last move was a capture and if we now statically evaluate the position without giving the opponent the opportunity to recapture, one will introduce a huge error in the evaluation. Therefore, usually all captures for both sides are played out before a position is statically evaluated.

More generally, regardless of the game, moves that have the potential of dramatically changing the static evaluation should be explored. By having the search only evaluate quiescence positions, the evaluation scores become more reliable.

Strictly speaking, this enhancement doesn't belong to the category of enhancements we present in this chapter. However, one can think of it as an extension to the evaluation function, and otherwise hidden from the search algorithm (in that case, we would need to pass the α and β search bound as arguments to *evaluate*(P)). Ever since the early days of computer chess the importance of searching these variations until quasi-stability is reached (before evaluating them) was recognized [74].

2.4 Best-First Search

The algorithms we described in this chapter traverse the game tree in a depth-first manner. That is, they fully explore each branch of the tree before turning their attention to the next. They all return the same minimax value; the primary difference is the search efficiency, where the more enhanced algorithms search a smaller tree (always at least the critical tree necessary for determining the minimax value is explored). There exists a different class of algorithms for searching game trees. These algorithms traverse the trees in a best-first fashion, and commonly search more selectively than depth-first methods. They temporarily stop exploring branches to visit other more interesting subtrees, possibly later returning to the abandoned branches to search them more deeply. However, these best-first algorithms are generally not time and space efficient and have therefore not found a wide use in practice. The best known of these algorithms are probably Stockman's *SSS** [77], Berliner's *B** [10, 11], McAllister's *Conspiracy Numbers* [53, 72] and Palay's *PB** [60]. We will not discuss these algorithms any further because we are only concerned with depth-

first search enhancements. For an overview of these alternative approaches interested readers can for example see Junghanns' review [43].

Chapter 3

Selective Depth-First Search

I look one move ahead ... the best!
- Siegbert Tarrasch

Although the term selective search has most often been associated with best-first search, the depth-first algorithms can also be selective in practice. The selectivity is introduced by varying the search horizon, some branches are abandoned prematurely, while others are searched beyond the nominal depth. The former case is referred to as *forward pruning* (we prefer the term *speculative pruning* as it is more descriptive), and the second as *search extensions*. Collectively, we refer to the two cases as *selective depth-first search*. As such, the search can return a value quite unlike that of a fixed-depth minimax search. In the case of speculative pruning, the full critical tree is not explored, and good moves may be overlooked. However, the rationale is that although the search occasionally goes wrong, the time saved by pruning non-promising lines is generally better used to search other lines deeper, i.e. the search effort is concentrated where it is more likely to benefit the quality of the search result.

3.1 Speculative Pruning

The real task when doing speculative pruning is to identify move sequences that are worth considering more closely, and others that can be pruned with

minimal risk of overlooking a good continuation. Several factors should be considered for effective pruning:

- *Risk assessment.*

  How safe is the pruning method? We want to minimize the risks that the speculative pruning introduces into the search.

- *Applicability.*

  To maximize the possible gains from pruning we would like to apply the method frequently in the tree, especially where there is a potential for big savings.

- *Cost effectiveness.*

  The investment of time and effort to decide whether to prune a node should be kept low. In any case, the savings achieved through pruning must exceed the additional effort introduced.

- *Domain dependency.*

  Ideally, we want a domain independent pruning method such that it can be applied in more than one specific game.

The above factors are by no means independent; improving one usually involves compromising another. For example, reducing the risks often means limiting the applicability, while improving cost effectiveness can introduce other risks. Finally the more general (domain independent) methods tend to be less efficient. A useful pruning heuristic must find the appropriate trade-off between the above factors, and this process may require careful tuning.

3.1.1 Risk Assessment

When using speculative pruning there is always some danger of overlooking good moves. We would like to minimize the risk of doing so. When deciding

whether to examine a node N, the basic question is: how likely is it that the subtree below N includes a continuation that, if searched, would yield a new principal variation. For a new variation to emerge two things must occur; first the value returned to N must exceed the best value found so far, and second the value must propagate to the root of the tree. This in turn implies that the pruning method should be able to:

- predict with reasonable accuracy the range of values for node N, and
- measure the likelihood that the anticipated value will back up to the root of the tree.

Existing speculative pruning methods address the first issue while ignoring the second one.

Error Introduction

For most subtrees, we are not so much interested in knowing the exact value of each particular node, but rather whether the value lies outside the bounds of the $\alpha\beta$ window. This is because we know that continuations that result in values outside the window can never become a part of the principal variation. When using a minimal-window search, the bound is the value of the current principal variation, so when comparing node values to the bound we are determining whether a better continuation is found. In that case we are simply interested in knowing if a value returned by searching a node further is at least as good as the β bound, since then it causes a cutoff.

When predicting where the value of a node N lies relative to the $\alpha\beta$ bounds, most pruning methods carry out a shallow search. They use the value returned to estimate the range in which the actual value of node N is likely to be found when the node is searched more deeply. For example, a 5-ply search is used to predict the outcome of a 6-ply search. The outcome of the shallow search

decides whether to search node N further. If we are confident enough that further search will not yield an improvement, node N is not expanded. The exact criteria used to relate the value of the shallow search to the anticipated return value of the deeper search varies with the pruning technique. Some approaches rely on statistical methods to define confidence intervals, while others simply use *ad hoc* heuristics. Error is introduced into the search when a wrong pruning decision is made.

Although values returned by shallow searches are usually reasonable estimates of the values found by deeper searches, additional information can enhance the overall prediction capabilities of the pruning heuristics, thereby reducing the risk involved. For example, consider the tree in Figure 3.1. The

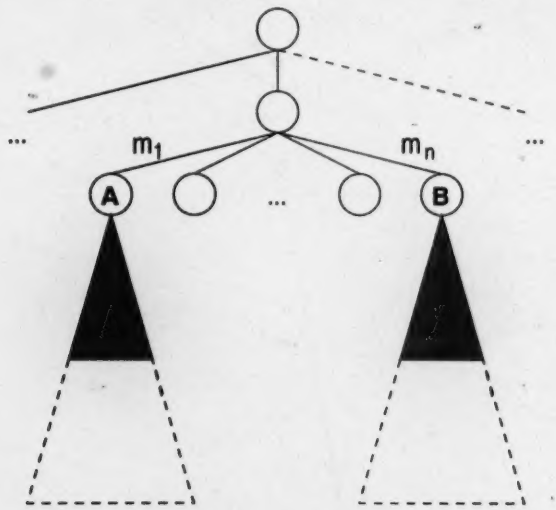


Figure 3.1: Different risk assessment of subtrees.

shaded area marks the parts of the tree searched to decide whether to prune nodes A and B. Each pruning decision is made independently, based only on the outcome of the local search. However, information is lost by looking at each node in isolation. For example, when looking at move m_n existing pruning methods are interested in knowing if the move will lead to a value that

causes a cutoff, that is, in estimating the probability

$$P(v(m_n) \geq \beta).$$

Having already searched moves $m_1,...,m_{n-1}$, and knowing that none of them caused a cutoff, provides a strong indicator that move m_n will also fail to do so, especially because the preliminary move-ordering scheme believes that move m_n is no better than the moves already considered. Instead one should compute the probability that move m_n causes a cutoff, given that moves $m_1,...,m_{n-1}$ have failed to do so. The probability can be expressed as:

$$P(v(m_n) \geq \beta \mid v(m_1), ..., v(m_{n-1}) < \beta).$$

That is, the values of the moves are not independent of each other and, by assuming so, otherwise potentially useful information is ignored. Existing pruning methods and probability-based best-first search algorithms ignore the dependencies, or unrealistically assume the search values (or the error in the values) are independent of each other. *Instead, the fact that the values tend to be dependent should be used to make more informed pruning decisions.*

Error Propagation

Figure 3.2 shows two different game trees. The solid lines identify the parts of the tree that have already been visited, while the dotted lines correspond to nodes that have not been expanded. Assume that the search is currently situated at node N and that the subtree resulting from playing move m_1 has already been searched. Furthermore, assume that a part of that subtree has been cut away using some speculative pruning technique, and that the value returned is greater or equal to the β bound for node N. Therefore, a β cutoff occurs and the value returned by move m_1 will back up to the root. From the root's perspective this branch is inferior to the current principal variation and

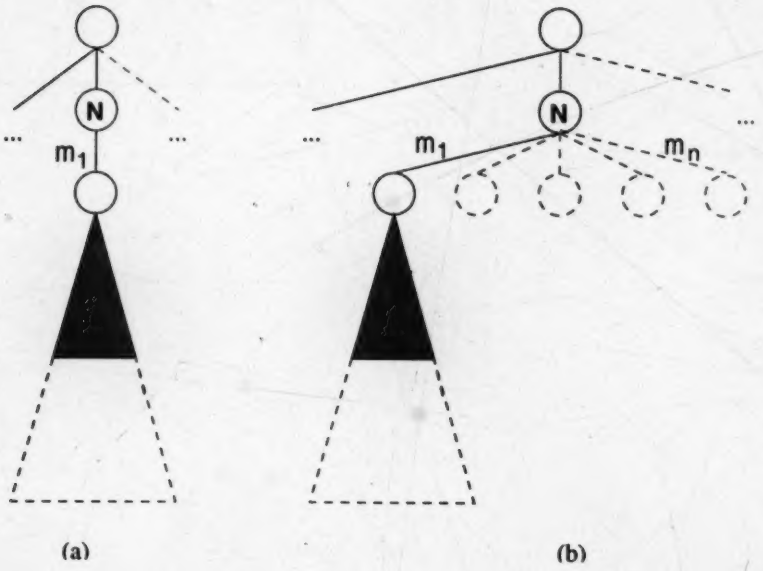


Figure 3.2: Different risk assessment of error propagation.

the search therefore continues to expand the other children of the root without switching principal variation.

If the pruned subtree in Figure 3.2(a) does not contain a better line, search effort has been saved. The case of interest here is: what if a better line *is* present? In Figure 3.2(a), if a better line is present but is overlooked, the value of m_1 is wrong and the error will propagate through node N to the root. However, if alternatives to m_1 are present, as in Figure 3.2(b), it is possible that one of the alternative moves in $[m_2, ..., m_k]$ may contain a line that enables it to deliver a β cutoff at N, acting as a substitute for m_1, and thus preserving the value assigned at node N. Thus in Figure 3.2(b), an error introduced by incorrectly pruning the subtree below m_1 does not necessarily propagate to the root. This situation is common in practice: if the first move fails to cause a cutoff, one of the alternative moves may do so. This means that even though the pruning below m_1 was flawed, the risk of affecting the move decision at the root is less in Figure 3.2(b) than in Figure 3.2(a), because

one of the other moves m_2 ..., m_n might preserve the cutoff if m_1 changes its value. Thus, *even though an erroneous pruning is made it will not necessarily affect the move decision at the root.* This illustrates that, when assessing risk, pruning methods should not only take into account the expected return value of a pruned node, but also assess the likelihood that an erroneous pruning decision will propagate up the tree.

3.1.2 Applicability

The most popular pruning heuristics used in two-person game-playing programs have one thing in common: they apply frequently throughout the search tree, though not without restriction. The more frequently a pruning heuristic is applied in the search, especially at places where there is a high probability of big savings, the more potential it has for being effective. However, the applicability is restricted, since pruning can only be done where it is expected to be safe. Depending on the heuristics used, this can differ substantially.

3.1.3 Cost Effectiveness

Although some pruning methods offer low risks and substantial savings in terms of nodes searched, the overhead needed to implement them is often prohibitive. The effort expended gathering and tracking in real-time the information required by the heuristics may outweigh the potential time savings introduced by the pruning. An example of such a heuristic is the *method of analogies*, a unique search reduction technique that was implemented in the pioneering KAISSA chess program [1]. Although, the method offers almost risk-free pruning, the overhead of tracking how pieces influence each other originally proved too high for practical use in a competitive chess playing program. However, changes in software and hardware technology may improve the viability of such methods. It might also be possible to approximate the

original heuristic by another that is less costly to maintain, and yet achieve most of the savings. Therefore the method of analogies is again a topic worthy of investigation.

3.1.4 Domain Dependency

Preferably we want domain independent pruning techniques. Those methods would not rely on such explicit knowledge as whether a king is in check, or whether a corner square is occupied. Instead the only information revealed to the search by the evaluation function is a numerical estimate of a problem state's quality. This clear separation of the search and the problem encourages more domain independent pruning methods. On the other hand the methods are then denied access to potentially useful information about the problem domain, thereby restricting their pruning capabilities. However, there is a wealth of information to be gathered about the problem by simply looking at the shape of the expanded search tree. This knowledge is accessible without having to uncover any additional domain-specific knowledge. We have already mentioned a few cases of interest as part of our risk-assessment discussion.

In practice, it is extremely difficult for pruning methods to be domain independent. As said earlier, there is a trade-off between generality and effectiveness, and to achieve the full pruning capability we must exploit some special characteristics of the search space. Most existing speculative pruning methods are therefore domain specific. Even though methods like *null-move* and *ProbCut* (see later) do not use explicit knowledge about their domain, they make implicit assumptions that tie them down for use in one, or at best very few, two-person games. For example, the null-move heuristic is very effective in chess, but inappropriate for Othello. Conversely, ProbCut is the pruning heuristic of choice in Othello but has not yet been shown useful in chess or checkers.

To support our claim that the new pruning method we introduce in a later chapter is indeed domain independent, we experiment with it using two different games as a test bed.

3.2 Search Extensions

The other side of selective search is search extensions. They are essential for improving decision quality of game-playing programs. For example, as mentioned in the introduction, one of the fundamental design decisions behind the search scheme employed by the DEEP BLUE chess-playing program was based around search extensions. However, most programs employ *ad hoc* domain dependent extensions schemes. For example, in chess it is common to extend forcing moves that have the potential of greatly altering the positional evaluation, such as checks, re-captures, and pushed of a passed pawn. In the past it has been a tedious and painful process to fine-tune the different extension schemes, because the decision quality of the search can degrade when extending too aggressively (because the search no longer reaches sufficient nominal depth). In a later chapter we introduce a novel method for automatically parameterizing search-extensions schemes. We will for most part postpone further discussion of search extensions until then.

3.3 Related Work

In this section we give a brief overview of the most popular selective search enhancements used by contemporary game-playing programs.

3.3.1 The Null-Move Heuristic

In some games, such as Go, a legal move is to pass (to make no move). The only change to the game state is the side to move. This is called a *null move.* In games like chess, the null move is not legal. Nonetheless, it can be useful in

the search to assume that a null move can be played. The idea of using null move in the search has been known for a long time [1], and is now used by most chess programs. However, the method did not get much attention in the literature until later [60, 5, 36, 6, 30].

When searching a position to depth d, before considering a legal move in that position a null move is made first. The position is then searched to a depth less than d, most often $d-1$ or $d-2$. If the resulting score is greater than β, a cutoff is made based on the shallow search result. The null move can be applied recursively in the tree. The underlying idea is that in chess it is almost always beneficial to make a move rather than to pass. Therefore, if the score received by giving up a move is still good enough to cause a cutoff, it is very likely that some of the legal moves will also cause a cutoff. Because the position was searched using a shallower search than we would otherwise, a considerable search effort is saved. In chess it is almost always safe to make the assumption that making a move will improve the position, but there are special cases in chess where this is not true. *Zugzwang* positions are the case in point, and are most likely to arise in the end game. Thus chess programs usually turn off the null-move heuristic when entering the end game. While the null-move heuristic works well in chess, it is useless in many other game-tree domains where zugzwang positions are common.

3.3.2 ProbCut and Multi-ProbCut

The *ProbCut* [24] heuristic uses shallow searches to predict the result of deep searches. In Othello, where the score of a position generally does not change significantly by searching deeper, this heuristic works very well. Therefore, if a shallow search predicts with a high confidence that a deeper search will produce a cutoff, a cutoff is made based only on the shallow search. A confidence interval of how good a predictor a shallow search is of a deeper search is

calculated off-line by searching a big database of positions that have been pre-classified into several different classes. A separate confident interval is calculated for each class. More recently, the method was further enhanced and the refined procedure named *Multi-ProbCut* [25].

3.3.3 Singular Extensions

In game-playing programs it is generally a good idea to search forced moves to a greater depth than other moves. Conventionally, search extension schemes rely on domain-specific knowledge to decide on the forcefulness of a move, for instance check evasions in chess. Another possibility, exploited by the *singular extension* heuristic [3], is to use the search itself to provide information about the forcefulness of a move. If a value returned by one move is significantly better than all others, that move is judged to be singular. Whenever a move is found to be singular, and it is likely to alter the outcome of the search if its value changes, the position arising from the move is re-searched one ply deeper. The idea of this search extension scheme is to allow the search to dynamically extend long forcing lines of play. Although the basic idea is simple, it requires extensive additional tuning and refining to get it to work smoothly in an actual game-playing program. More recently this method has been further refined, for example to extend not only on singular moves but also binary, and trinary moves (i.e. only 2 or 3 good moves). The new refined scheme was successfully employed by DEEP BLUE [27].

3.3.4 Other Methods

In the early days of computer chess there was interest in speculative pruning methods. Today, most of these methods are of a limited practical use. Although not in common use today, some of these methods were quite novel (e.g. the aforementioned method of analogies [1]). Later some of these early

speculative pruning methods further evolved and became popular, for example *razoring* [12] and *futility pruning* [75, 70]. When the null-move pruning technique became mainstream it superseded some of the earlier pruning methods. In the last decade there has been revived interest in speculative pruning methods. Preliminary experimental results with a method named *Fail-High Reductions* have been reported [32], however, they didn't offer much additional benefit when used alongside established pruning methods such as null-move pruning, thus limiting its usefulness. More recently, *AEL* pruning was introduced [39]. This method is a collection of three pruning schemes: *adaptive null-move pruning*, *extended futility pruning*, and *limited razoring*; each enhancing an older existing pruning scheme. It showed great promise in the chess program DARK THOUGHT.

3.4 Conclusions

We have given an overview of existing selective search methods, and pinpointed some of their short-comings. As mentioned before, speculative pruning heuristics should be concerned with the question: *What is the likelihood of making an erroneous pruning decision and, if an erroneous decision is made, how likely is it to affect the principal variation?* The existing methods generally do not consider the second part of this question. When assessing risk, pruning methods should not only speculate whether a subtree contains a good continuation, but also determine if there are alternatives to any potentially overlooked continuation that could preserve the principal variation. To answer these questions the methods must consider each node in the context of its location in the game tree, instead of looking at each node (and the subtree below it) in isolation.

In a later chapter we present a new speculative pruning method that overcomes some of the aforementioned shortcomings.

Chapter 4

Uncertainty Cutoffs

Of chess it has been said that life is not long enough for it, but that is the fault of life, not chess."
-William Ewart Napier

In this chapter, we take a new look at pruning. A common scenario in a search is that expectations change. Uncertainty in the search results in changes of the principal variation (PV). When this happens, some branches are explored that, with hindsight, are unnecessary. There is an opportunity here for savings. A new pruning technique, *uncertainty cutoffs*, is applied at carefully selected places in the search tree. Bookmarks are kept where the pruning is done, so that one can tell if a backed-up value is a correct minimax value or an *uncertain* value. Even if speculative pruning is used in the search, it does not necessarily affect the reliability of the minimax value at the root of the search tree. If the pruning is only done in subtrees that turn out to be irrelevant for proving the minimax value, a guaranteed value can be backed up to the root. The bookmarks tell us whether the pruning applied in the tree is affecting the reliability of the minimax value, thus giving us the opportunity to correct it by re-searching the subtrees containing the uncertain values. Hopefully, the gains of applying the pruning will outweigh the extra search overhead of occasional additional re-searches of uncertain values.

Strictly speaking, this pruning method does not fall into the category of speculative-pruning methods because we have a way of telling when a value is uncertain and can rectify the situation by re-searching the node. The type of pruning introduced here is analogous to the additional pruning power introduced by minimal-window searches, where an artificial upper bound is used: if the search fails-low the gamble pays off, but if the search fails-high a re-search is necessary.

In the next section we take a second look at the critical tree that must be searched to prove the value of a game tree. This is followed by two sections that describe the uncertainty cutoff pruning method, the idea and the implementation, respectively. Finally, we present our assessment and provide experimental results.

4.1 Searching a Critical Tree

We have previously mentioned that to find the value of a game tree, at least a so-called *critical tree* must be searched. Here we take a closer look at the structure of a critical tree. The nodes of a critical tree can be categorized into three different types based on their properties, as shown in Figure 4.1. The light colored nodes in the picture belong to a critical tree. All moves have to be searched at pv-nodes (P) and all-nodes (A), but only one move is searched at cut-nodes (C). The dark colored nodes need not be searched, but some of them may be, depending on the quality of the move-ordering scheme used. Before searching a node we do not really know if a node will become a cut-node or an all-node. Thus, before fully exploring nodes we refer to them as *expected cut-nodes* or *expected all-nodes* depending on if we believe they will cause a cutoff or not.

The performance of the $\alpha\beta$ algorithm is affected by the order in which nodes in the tree are searched. In the best case only a critical tree is expanded. The

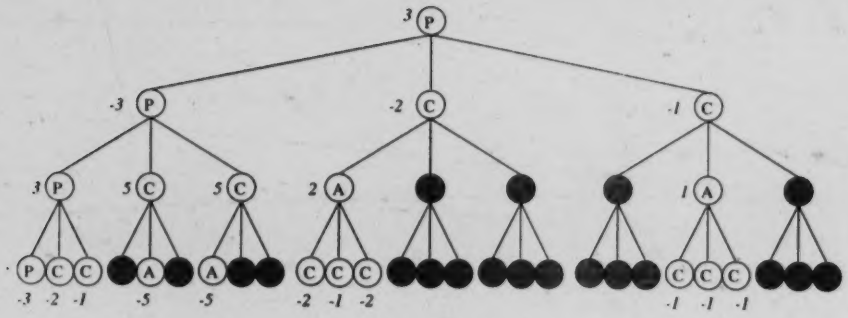


Figure 4.1: Critical tree.

best move must be expanded first at pv-nodes (to get a good lower bound early), but at cut-nodes any move sufficiently good to cause a cutoff can be searched first.[1] Because of the $\alpha\beta$ algorithm's sensitivity to the move ordering, it is important to expand good moves as early as possible. Various heuristics to achieve good move ordering have been developed in the past (see Chapter 2). By using these heuristics in chess, for example, empirical evidence shows that over 90% of the cases where a cutoff occurs it is indeed caused by the first move. As mentioned in Chapter 2, enhanced $\alpha\beta$ variants like NegaScout and Principal Variation Search that employ minimal-window search take advantage of moves that are ordered such that good ones are more likely to be searched first. These algorithms have been shown, both theoretically [65] and empirically [70], to be more efficient than the original $\alpha\beta$ algorithm.

4.2 Uncertainty Cutoffs - Idea

Current tree-search algorithms equipped with various search enhancements are searching quite efficiently. But there is still scope for improvement. Search overhead from imperfect move ordering can be introduced in two ways:

[1] Because of a non-uniform branching factor, search extensions and various possible transpositions, the size of subtrees generated by different moves may vary considerably. In general, we would like to search first not only a move that returns a value that is sufficient to cause a cutoff, *but* also one that leads to the smallest subtree.

- at a cut-node, the first move does not cause a cutoff, or
- at a pv-node, the first move is not the best.

Both these cases occur when there is uncertainty in the search — previous expectations are changing. In the first case additional moves must be searched until a move (if any) causes a cutoff. The subtrees of the sibling nodes searched prior to the node that caused the cutoff have been searched unnecessarily. In Figure 4.2 this search overhead is shown at node b as the shaded subtree T_1. In the second case, assuming minimal-window search is used, when a new best move is found it must be re-searched with a normal window. The search overhead here consists primarily of the initial minimal-window search that failed high.[2] Figure 4.2 shows the case when the third move searched at the root (i.e. c) fails high; the minimal-window search that is performed (the shaded subtree at c) is the search overhead and the subtree T_2 represents the necessary re-search. However, information stored in the transposition table during the minimal-window search efficiently guides the re-search, saving some move generations and node expansions.

At cut-nodes it is most important that the move which causes the cutoff be searched as soon as possible. To improve the prospect of choosing a move that will cause a cutoff, we make use of available move information (e.g. the transposition table entry and the history heuristic). However, *while searching the subtree of this move we might, based on other information, start to believe that this move will not cause a cutoff.* The question that we then face is whether to continue searching this sub-branch, or to stop and start searching a different candidate cutoff move nearer the root of the tree. This is the basic idea behind the pruning method introduced here. Instead of having only the two scenarios (either expanding all children of a node or having a child

[2]The search efficiency is also somewhat degraded because prior sibling nodes have been searched with an inappropriate window.

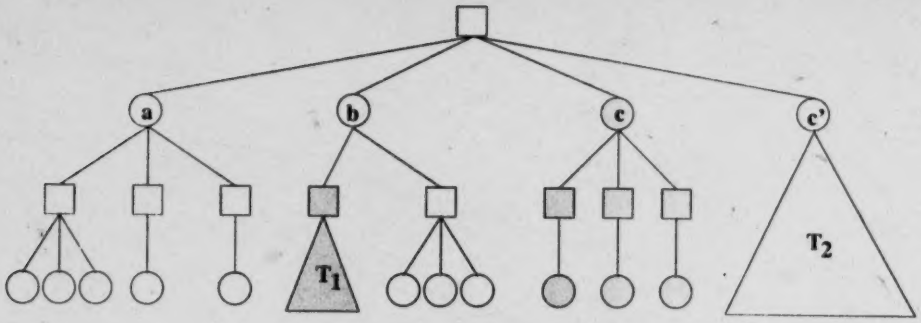


Figure 4.2: Search overhead.

cause a cutoff) a third scenario is now also possible, where only some of a node's children are searched before we stop. This type of pruning shows some resemblance to the additional pruning possible when using a minimal-window search. Both uncertainty cutoffs and minimal-window search allow speculative cutoffs based on the expectation that an alternative move nearer the root of the tree is more likely to cause a cutoff than the move currently being explored. The difference is that uncertainty cutoffs make a speculative cutoff at expected cut-nodes (if they do not produce a cutoff quickly), whereas the artificial upper bound of a minimal-window search enables early cutoffs at expected all-nodes (if their lower bound happens to exceed or equal the artificial bound).

To illustrate this idea in practice, let us look at the chess position in Figure 4.3. Here it is White's turn to move. The pawn on *e*5 is threatened but White has several possible continuations. Assume that White has already found a tentative principal variation and is now thinking of 1.*e*6 as an alternative move. Black's obvious reply 1....*R*×*e*6 fails to 2.*N*c5, attacking both of Black's rooks. That threat, however, was beyond the search horizon of the previous search iteration, so the search expands the move 1....*R*×*e*6 first and White responds with 2.*N*c5 (not necessarily the first move considered). In the resulting position Black is faced with the problem of saving the rooks. Black has over

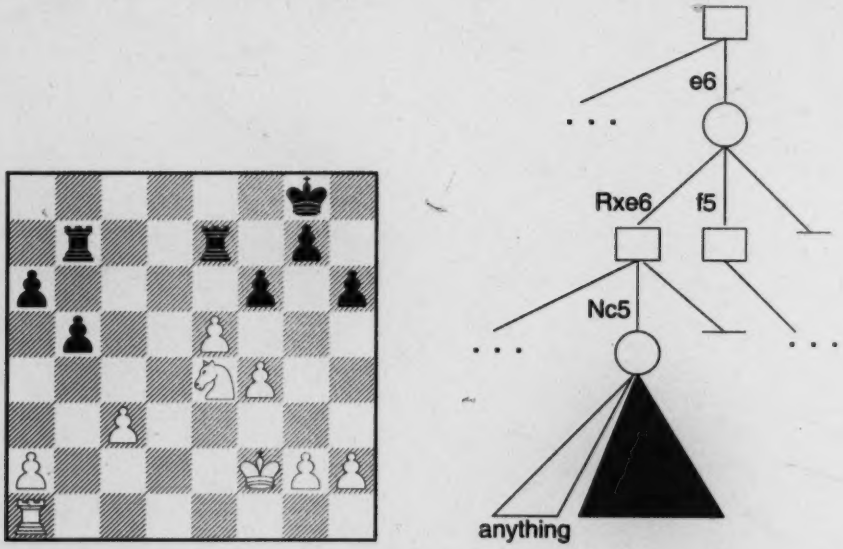


Figure 4.3: Example position and a corresponding search tree.

25 legal moves but all fail to prevent White from capturing one of the rooks. Instead of exhaustively searching all the possible legal moves, we can abandon the others after only a few have been examined, and start to look at alternatives to 1....$R\times e6$. A better move is easily found (e.g. 1....$f5$) and, assuming the new move generates a cutoff, then we save considerable search effort, but still return the same, correct minimax value. The search tree corresponding to this example is also shown in Figure 4.3. This tree is expanded during a minimal-window search, where the shaded area represents the part of the tree not searched because of the uncertainty cutoff. The savings arise because the sequence $e6, R\times e6; Nc5$, "*anything*" looks like a new principal variation. Rather than exploring all the alternatives for "anything" we assume that a new principal variation is indeed emerging and so retreat up the tree, where we quickly refute the candidate principal variation with $f5$.

4.3 Uncertainty Cutoffs - Algorithm

Two fundamental questions must be considered when implementing the above pruning method: how to guarantee that a correct minimax value is returned, and how to make decisions about when to apply the pruning method. These questions will now be addressed.

Assume that the pruning method described above is applied in subtree T_1 of Figure 4.2, such that the value backed up to the root of T_1 is not guaranteed to be the correct minimax value, i.e. the value is uncertain. The interesting case occurs when the value returned by T_1 does not cause a cutoff, but another child of b does. In that case the subtree T_1 is not a part of the critical tree and any pruning made in there will not affect the true value in any way. Given that this value of the move causing the cutoff is not uncertain, then neither will the value returned by b. However, in cases where b fails low the value returned by b will be uncertain if *any* of its children's values are uncertain. If an uncertain value is backed up all the way to a pv-node, that node will have to be re-searched. By keeping track of how uncertain values are backed up in the tree, we can determine if the returned value for the search is guaranteed to be the correct minimax value or not.

Below we show how uncertainty-cutoffs can be embedded into the *Principal-Variation Search* algorithm. Algorithms 9 and 10 show how we need to modify the PVS and MWS functions, respectively. In the $ucPVS$ (uncertainty cutoff PVS) function line 14 has been modified to have the minimal-window search return if a backed-up value is uncertain, and lines 15-16 is the additional code to handle the re-search of uncertain values. Note that the returned value cannot even be used as a bound for the re-search. Otherwise the function is the same as the original PVS function. When an uncertain value is backed up to a pv-node, its value is corrected by re-searching that node using the current

Algorithm 9 $ucPVS(P, d, \alpha, \beta)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: M ← generateMoves(P)
5: make(P, m_1)
6: best ← −ucPVS(P, d − 1, −β, −α)
7: retract(P, m_1)
8: if best ≥ β then
9:   return best
10: end if
11: lower ← max(α, best)
12: for all m_i ∈ M|i > 1 do
13:   make(P, m_i)
14:   (v, uncertain) ← −ucMWS(P, d − 1, −lower, true)
15:   if uncertain then
16:     v ← −ucPVS(P, d − 1, −β, −lower)
17:   else if v > lower and v < β then
18:     v ← −ucPVS(P, d − 1, −β, −v)
19:   end if
20:   retract(P, m_i)
21:   if v > best then
22:     best ← v
23:     if best ≥ β then
24:       return best
25:     end if
26:     lower ← max(α, best)
27:   end if
28: end for
29: return best
```

$\alpha\beta$ window. Therefore there is no need to keep track of uncertain values in the PVS part of the algorithm. Thus the method represents a "safe" pruning mechanism.

The uncertainty cutoffs themselves take place in the $ucMWS$ function, and uncertainty information about a value is backed up there. Basically, a backed-up value is uncertain at all-nodes if at least one of its values is uncertain (the variable *uncertain* is used to record this). On the other hand, at cut-nodes the returned value is uncertain if and only if the move that caused the

Algorithm 10 $ucMWS(P, d, \beta, cut)$

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: uncertain ← false
5: best ← −∞
6: M ← generateMoves(P)
7: for all m_i ∈ M do
8:   make(P, m_i)
9:   (v, uc) ← −ucMWS(P, d − 1, −β + ε, ¬cut)
10:  retract(P, m_i)
11:  if v > best then
12:    best ← v
13:    if best ≥ β then
14:      return (best,uc)
15:    end if
16:  end if
17:  if cut and doUncertaintyCut(d, m_i, M) then
18:    return (best,true)
19:  else if uc then
20:    uncertain ← true
21:  end if
22: end for
23: return (best,uncertain)
```

cutoff has an uncertain value (the variable *uc* tells if the value of the move just searched is uncertain). The implementation of the specific strategy for deciding exactly when to apply the uncertainty cutoffs may differ somewhat between games, thus we abstract it here in the $doUncertaintyCut(d, m_i, M)$ function (we discuss some strategies a little later). Furthermore, we must be careful to specially mark uncertain nodes when inserting them into the transposition table, so that their re-use is restricted to suggesting the best move, and not to adjusting the search bounds or the search value.

The other fundamental question is where and when to apply the pruning heuristic. We cannot blindly apply the pruning everywhere in the tree, because this would result in frequent re-searching of uncertain nodes, resulting in the search overhead of the re-searches exceeding the gains of the pruning.

Basically, we would like to prune only in subtrees that are not likely to become a part of the minimal tree. What is needed is a good criteria for identifying these subtrees. Typically what happens is that the shape characteristics of the search tree change when we are searching on a path that is off the critical tree. Nodes that we expect to be cut-nodes start to behave like all-nodes and vice versa. This can be seen in Figure 4.2. For move c to fail high (and therefore is no longer a part of the critical tree because of the re-search) it must be true that all children of c are searched and fail low. Because of the move ordering, the moves that are most likely to cause a cutoff are examined first, but if none of the promising cutoff candidate moves causes a cutoff, we have good reason to believe that the rest of the moves will also fail to do so. Therefore, after searching only some of the possible moves at c we may decide not to search the rest, i.e. we make an uncertainty cutoff, and return right away. This will cause node c to be re-searched. The criteria used here to decide when to apply the pruning is as follows: *if during a minimal-window search, a node that is expected to be a cut-node does not cause a cutoff after searching some number of moves, then the rest of the moves are ignored.* The number of moves looked at in each position can be determined in various ways; for example a fixed percentage of legal moves could be searched (or possibly a different more dynamic measure). Also, because we only test for uncertainty cutoffs if a node is an expected cut-node, an extra parameter *cut* is passed down to the $ucMWS$ function. It is set to true if the node we are currently visiting is an expected cut-node, but is otherwise false. In a minimal-window search there are alternating layers of cut- and all-nodes (recall our discussion of the critical tree), thus the value of the *cut* parameter is negated in each recursive call.

4.4 Experimental Results

The method was implemented in THE TURK, an experimental chess program.[3] The program's search engine uses the Principal-Variation Search algorithm (PVS/MWS). It also employs most search enhancements found in contemporary chess programs. The move-ordering scheme generates capture moves first (most valuable piece captures generated first) and the history heuristic is used to sort the remaining moves. The best move previously found in a position is stored in the transposition table and searched first where applicable.

For our experiments we used the test positions published in Plaat's PhD thesis [63] (the positions are also listed in Appendix D.1). The test positions were searched to a depth of 8-ply but, as in the previous mentioned work, with both search extensions and pruning enhancements disabled.[4] On the other hand, enhancements such as transposition tables, iterative deepening and quiescence search were used. Uncertainty cutoffs are done at expected cut-nodes after a fixed percentage of the legal moves are searched, with the proviso that capture moves are always searched. Furthermore, the pruning is not done if the remaining search depth is less or equal to one, because at these frontier nodes we have poor move-ordering information and, consequently, the best move can lie almost anywhere in the move list.

For different parameter values, the graphs in Figure 4.4 show how the uncertainty-cutoff program performs relative to the same program without the cutoffs. The moves-looked-at ratio — the percentage of moves explored at expected cut-nodes before making an uncertainty cutoff — is varied from 10-90% (the x-axis). The graph to the left shows the number of nodes searched by the uncertainty-cutoff version compared with the unmodified program, whereas

[3] THE TURK was developed at the University of Alberta by Yngvi Björnsson and Andreas Junghanns.

[4] Using these selective search techniques would make it difficult to measure the search efficiency, because radically different trees are possibly expanded from one run to the next.

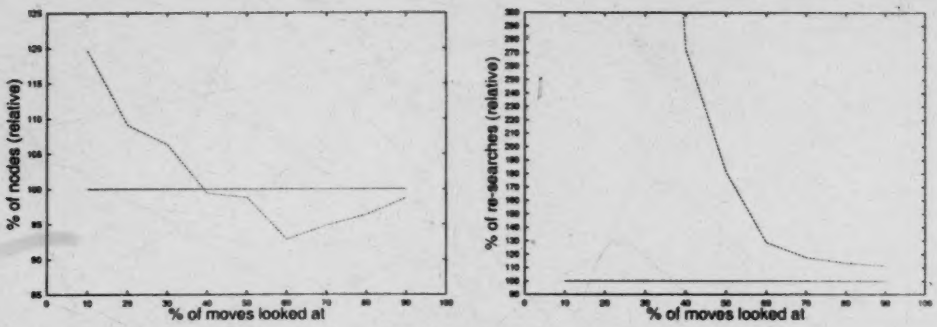


Figure 4.4: Efficiency comparison using the Plaat test positions.

the graph to the right shows the relative number of re-searches.

As can been seen from the graphs, when we apply the pruning too aggressively the total number of nodes searched is higher than nodes seen without the pruning. The reason is that too many incorrect pruning decisions are made, resulting in frequent re-searches to correct the uncertain minimax values. This is even more apparent when looking at the node-count information in conjunction with the re-search information, presented in the graph to the right; when only a small fraction of possible moves is explored, exponentially more re-searches are performed. However, as the moves-looked-at ratio increases, the number of additional re-searches drops rapidly. Also, more importantly, so does the number of total nodes searched. The two versions of the program break even at the 40% mark, from which point on the pruning "gamble" starts to pay off, that is, the uncertainty-cutoff version now searches fewer nodes in total. Clearly, the savings gained by pruning some of the sub-branches more than outweighs the extra search overhead introduced by the occasional re-searches. As the moves looked at ratio approaches 100%, the two versions converge to identical behavior as one would expect. Even though the data here are presented as savings in nodes searched, the run-time overhead with the method is negligible so the search time results are in the same ratio.

Table 4.1 shows the search efficiency broken down by individual positions in

Table 4.1: Uncertainty cutoff results on the Plaat test positions.

	moves-looked-at ratio								
Pos.#	0.1	0.2	0.3	0.4	0.5	**0.6**	0.7	.8	0.9
1	115.84	114.71	109.68	100.78	105.88	99.10	101.04	102.31	108.48
2	129.46	131.56	129.60	110.94	102.49	97.54	97.00	98.70	100.59
3	118.92	113.76	110.65	111.23	98.71	92.66	96.40	97.82	99.04
4	109.20	96.19	94.10	79.21	92.43	79.05	87.99	92.56	98.50
5	110.43	109.94	103.21	92.50	93.22	89.75	93.99	98.00	100.97
6	121.70	116.10	113.15	93.89	89.85	91.17	99.50	100.14	101.19
7	114.83	106.94	101.70	101.19	100.08	100.10	99.90	99.68	99.95
8	96.47	92.24	95.44	88.03	83.22	83.89	86.26	88.75	93.33
9	131.58	117.48	111.23	98.40	95.34	95.38	99.67	100.32	100.56
10	132.69	114.89	133.60	126.14	116.20	105.03	96.07	97.85	98.80
11	126.44	132.45	139.92	151.30	149.97	96.28	97.82	97.31	98.49
12	127.78	100.50	100.26	100.09	94.34	95.32	97.90	97.20	98.46
13	129.18	97.05	95.10	81.70	92.81	86.21	89.21	92.00	95.34
14	141.12	119.19	96.91	89.49	96.15	90.66	92.89	94.81	97.10
15	134.72	112.25	108.04	98.12	98.50	98.40	98.66	98.98	99.47
16	110.11	106.40	103.48	99.77	100.81	99.81	99.88	99.93	99.99
17	116.98	118.02	103.46	103.27	103.24	102.58	99.84	99.85	99.92
18	104.75	90.03	87.20	84.27	80.12	85.87	86.93	89.99	94.82
19	134.93	121.19	103.79	107.62	101.51	99.94	99.45	99.28	99.52
20	131.57	110.30	104.64	99.42	104.66	96.10	97.98	97.30	98.83
Tot.%	119.68	109.10	106.33	99.46	98.82	**92.87**	94.87	96.36	98.78
Avg.%	121.94	111.06	107.26	100.87	99.98	**94.24**	95.92	97.14	99.17

the test suite. The table entries show the percentage of nodes searched relative to an unmodified program. The total savings at the bottom of the table are shown both as a percentage of nodes searched in total, as well as the average percentage saving over all the positions (i.e. the percentage numbers shown in the table). It is interesting to note that for the optimal moves-looked-at ratio, in only few cases does the pruning cause more nodes to be searched, and then only marginally. On the other hand, the savings can be substantial.

As a further proof of concept, we ran identical set of experiments using a different test suite, the so-called Bratko-Kopec positions (see Appendix D.2). The performance graphs are produced in Figure 4.4, and yield similar results.

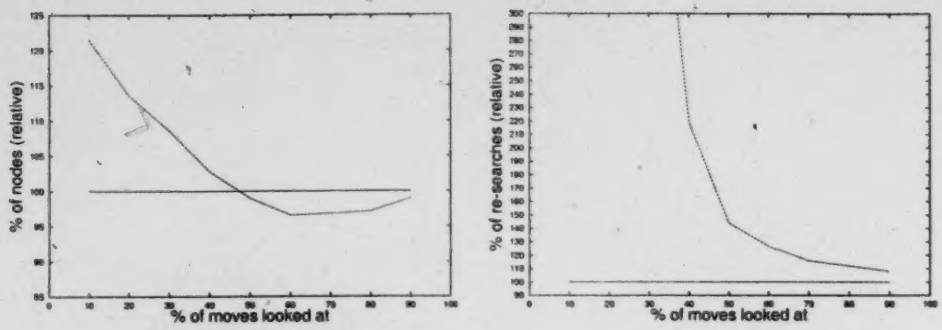


Figure 4.5: Efficiency comparison using the Brato-Kopec test suite.

4.5 Conclusions

We introduced a new enhancement that improves the search efficiency of the Principal-Variation Search algorithm (and other $\alpha\beta$-like algorithms), while still backing up the correct minimax value. However, because move ordering in chess programs is already very good, and programs are searching quite close to the critical tree (needed to prove the minimax value), the savings are necessarily relatively small. In other less researched game domains, where good move-ordering information is not as easily available, this enhancement may offer additional savings. Nonetheless, the efficiency improvement the new enhancement yields is comparable to what other similar search variants demonstrate, e.g. the efficiency gains NegaScout/PVS shows over $\alpha\beta$, or MTD(f) over NegaScout [63]. Thus, we view this new technique as yet another important step in a long line of algorithmic enhancements that aid $\alpha\beta$-based search in achieving close to optimal search behavior.

Improvements that only aim at improving the search efficiency, while still insisting that the correct minimax value be proved, will yield only marginal improvements. Therefore, we will shift our focus to speculative search enhancements that do not necessarily prove the minimax value, but instead aim at improving the overall decision quality of the search.

Chapter 5

Multi-Cut $\alpha\beta$ Pruning

When you see a good move – wait – look for another.
- Emmanuel Lasker

This chapter introduces a new speculative pruning enhancement to the $\alpha\beta$ algorithm. It is based on our earlier observation that pruning methods should not only consider the likelihood that a subtree contains a better continuation, but also how likely it is that an erroneous pruning decision will propagate back up the tree to influence the move decision at the root (see Chapter 3 for a more complete discussion).

5.1 Multi-Cut Idea

In the traditional $\alpha\beta$-search, if a cutoff occurs there is no reason to examine that position further, and the search can return. For a new principal variation to emerge, every expected cut-node on the path from a leaf-node back to the root must become an all-node. In practice, however, it is common that if the first move does not cause a cutoff at an expected cut-node, one of the alternative moves will. Therefore, *expected cut-nodes, where many moves may have good potential for causing a β-cutoff, are less likely to become all-nodes. Consequently such lines are unlikely to become part of a new principal variation.* This observation forms the basis for the new speculative pruning

scheme we introduce here, called *multi-cut $\alpha\beta$-pruning*. Before explaining how it works, let us first define an *mc*-prune (multi-cut prune).

Definition 1 (*mc*-prune) *When searching node N to depth $d+1$ using an $\alpha\beta$-like search, if at least c of the first e children of N return a value greater than or equal to β when searched to depth $d-r$, an mc-prune is said to occur and the local search returns.*

Figure 5.1 illustrates the basic idea. At node N, before searching move m_1 to its full depth d, like a normal $\alpha\beta$-search will, the first e successors of N are expanded to a reduced depth of $d-r$. If c of them return a value greater than or equal to β, an *mc*-prune occurs and the search returns the β value, otherwise the search continues as usual exploring move m_1 to the full depth d. The moves $m_2, ..., m_e$ are searched to depth $(d-r)$ and represent the extra search overhead introduced by *mc*-prune. This overhead would not be incurred by normal $\alpha\beta$-search. On the other hand, the dotted area of the subtree resulting from move m_1 represents the savings that are possible if the *mc*-prune is successful. However, if the pruning condition is not satisfied, we are left with the overhead but no savings. Clearly, by searching the subtree of move m_1 to a shallower depth, there is some risk of overlooking a tactic that would result in m_1 becoming a part of the new principal variation. We are willing to take that risk, because we expect at least one of the c moves that return a value greater or equal to β when searched to a reduced depth, would have caused a genuine β-cutoff if searched to a full depth, d. If a shallow search returns a value good enough to cause a cutoff, that is generally a strong indicator that a deeper search will also cause a cutoff. The fact that we require more than one such shallow cutoff further reduces a risk of erroneous mc-pruning.

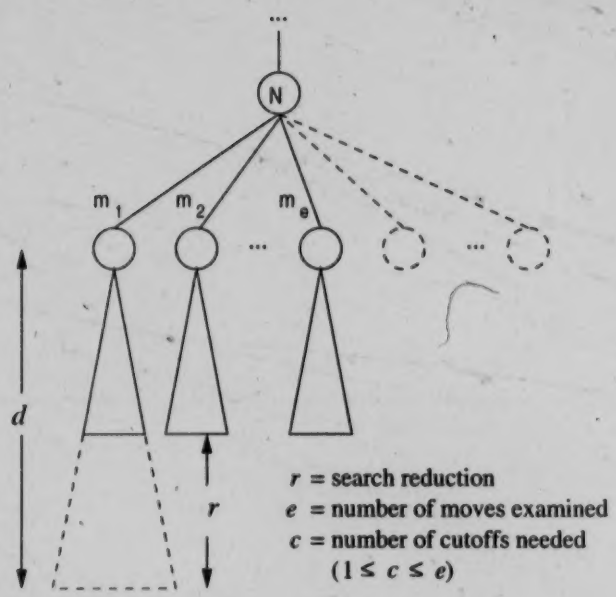


Figure 5.1: Applying the *mc*-prune method at node N.

5.2 Multi-Cut Implementation

Algorithm 11 lists the pseudo-code for a minimal-window search (MWS) routine using multi-cut. The MWS routine is an integral part of the Principal Variation Search algorithm (see Chapter 2). The multi-cuts are applied only in the NWS routine, whereas the PVS routine is unchanged.

Multi-cut could equally well be implemented in a standard $\alpha\beta$ algorithm or a different variant (e.g. *NegaScout*). For clarity we have omitted details about search extensions, transposition table look-ups, quiescence searches, null-move searches, and history heuristic updates that are irrelevant to our discussion. The parameter d is as before the remaining length of search for the position, and β is an upper bound on the value we can achieve. The new parameter, *cut*, is set to true if the node we are currently visiting is an expected cut-node, but is otherwise false. In a minimal-window search there are alternating layers of cut- and all-nodes, thus the value of the *cut* parameter is negated in each

recursive call (information from the transposition table can also be used to tell if we expect a node to be a cut-node or not).

The routine starts by checking whether the search horizon is reached, and if so evaluates the position and returns its value (or calls a quiescence search). If we are using a fully enhanced search routine, we would next look for useful information about the position in the transposition table, followed by a null-move search. If the null-move does not cause a cutoff, a standard minimal-window search would follow (lines 20–30). However, when using multi-cut, the check for the pruning condition is inserted before we start exploring the possible moves (lines 6–19). The parameters E, R, and C are *mc*-prune specific and stand for: number of moves to expand (e), search reduction (r), and number of cutoffs needed (c), respectively (see Figure 5.1).

We do not check for the *mc*-prune condition at every node in the tree. First, we test for the condition only at expected cut-nodes (we would not anticipate it to be successful elsewhere). Second, multi-cut is not applied at levels of the search tree close to the horizon, thus reducing the time overhead involved in this method. We experimented with distances both closer to and further away from the horizon, and a distance of R gave a good balance. Finally, there are some game-dependent restrictions that apply, but are not shown in the pseudo-code. In our experiments in the domain of chess (see later) the pruning is disabled when the end-game is reached, since there are usually few viable move options there and the *mc*-searches are therefore not likely to be successful. Also, the positional understanding of chess programs in the end-game is generally poorer than in the earlier phases of the game. The programs rely more heavily on the search to guide them in the ending, and any speculative-pruning scheme is therefore more likely to be harmful. Furthermore, the pruning is not done if the side-to-move is in check, or if search extensions have been applied for any of the three previous moves leading

Algorithm 11 $mcMWS(P, d, \beta, cut)$

Require:

E is the number of moves to expand when checking for *mc*-prune.
C is the number of cutoffs to cause an *mc*-prune.
R is the search depth reduction for *mc*-prune searches.

```
1: if d ≤ 0 or isTerminal(P) then
2:   return evaluate(P)
3: end if
4: best ← −∞
5: M ← generateMoves(P)
6: if d ≥ R and cut then
7:   c ← 0
8:   for m_i ∈ M | i = 1, ..., E do
9:     make(P, m_i)
10:    v ← −mcMWS(P, d − 1 − R, −β + ε, ¬cut)
11:    retract(P, m_i)
12:    if v ≥ β then
13:      c ← c + 1
14:      if c = C then
15:        return β
16:      end if
17:    end if
18:  end for
19: end if
20: for all m_i ∈ M do
21:   make(P, m_i)
22:   v ← −mcMWS(P, d − 1, −β + ε, ¬cut)
23:   retract(P, m_i)
24:   if v > best then
25:     best ← v
26:     if best ≥ β then
27:       return best
28:     end if
29:   end if
30: end for
31: return best
```

to the current position. The presence of extensions usually indicates forced continuations, and pruning in such cases is often risky.

5.3 Multi-Cut Parameters

It is not clear how to select the most appropriate values for the parameters c, e, and r. How they are set will affect both the efficiency and the error rate of the search. Each parameter influences the search in its own way:

- *Number of cutoffs (c):*

  The more cutoffs that are required for an *mc*-prune to occur, the safer the method is. On the other hand, the higher the value is, the larger the tree expanded. Not only does each check for an *mc*-prune require more nodes to be searched, but also fewer *mc*-prunings occur. Therefore, c should be set large enough for the method to be reasonably safe, but still small enough to offer substantial node savings.

- *Number of moves (e):*

  The e parameter tells how many moves to investigate when checking for an *mc*-prune. The higher e is, the more likely it is that the pruning condition will be met. On the other hand, each unsuccessful *mc*-prune search is a failed investment of search effort, offsetting some of the node savings from the additional pruning. The right balance between these two counter-acting effects will depend, among other things, on the quality of the move ordering scheme used. The better the scheme, the closer we can set e to c.

- *Depth reduction (r):*

  The depth reduction factor r will influence the best settings for c and

c; the larger r is, the larger c and e can be. Obviously, if the goal is to improve search efficiency, the depth reduced multi-cut searches must explore, in total, fewer nodes than the full depth search they replace. Therefore, if r is very small there is not much flexibility in choosing larger values for c and e. On the other hand, search depth reduction that is too aggressive will make the search more error-prone.

From the above discussion we can see how intertwined the parameters are; altering one will bias the selection of the others. It is impossible to analytically determine the most appropriate settings for the parameters, because not only do they depend on different characteristics of the search space, but also on various properties of the game-playing program itself (e.g. the move-ordering scheme). We empirically determined a suitable setting of these parameters for our experiments (see next section).

5.4 Experimental Results

To test the idea in practice, multi-cut $\alpha\beta$-pruning was implemented in two different game-playing programs: first the chess program THE TURK, and more recently in a Lines-of-Action program named YL. Three different kinds of experiments were conducted using the chess program. First, the feasibility of the idea was verified by correlating the number of promising move alternatives at cut-nodes to an actual cutoff occurring. Secondly, different multi-cut parameter settings were experimented with both to give insight into how they alter the search, and to find an appropriate setting for the chess program. Finally, the improved playing strength of programs using the new pruning scheme was demonstrated via self-play matches, for both chess and Lines of Action.

5.4.1 Criteria Selection

The multi-cut idea stands or falls with the hypothesis that nodes having many promising move alternatives are more likely to cause a β-cutoff than those with few. Before arriving at the multi-cut implementation described earlier, we tested several different schemes of predicting which nodes are likely to deliver a β-cutoff. We will refer to any node where a β-cutoff is anticipated as an expected cut-node. Only after searching the node do we know if it actually causes a cutoff; if it does we call it a *True-cut-node*, otherwise a *False-cut-node*. What we seek is a scheme that accurately predicts which expected cut-nodes are False. We experimented with four different ways of anticipating cut-nodes:

1. Number of legal moves (NM):

   The most straightforward approach is to assume that every move has the same potential for causing a β-cutoff. Thus, the more children an expected cut-node has, the more likely it is to be a True-cut-code. Although this assumption is not realistic, it serves as a baseline for comparison.

2. History heuristic ($HH > \Delta$):

   A more sensible approach is to distinguish between good and bad moves, for example by using information from the history-heuristic table. Moves with a positive history-heuristic value are known to be useful elsewhere in the search tree. This method defines moves with a history-heuristic value greater than a constant Δ as potentially good. One advantage of this scheme is that no additional search is required.

3. Quiescence search ($QS() \geq \beta - \delta$):

   Here quiescence search is used to determine which children of a cut-node have the potential for causing a cutoff. If the quiescence search returns a value greater than or equal to $\beta - \delta$ then the child is considered

promising. The constant δ, from here on called the β-cutoff margin, can be either positive or negative. Although this scheme may require additional search, it is likely to give a better estimate than the two aforementioned schemes.

4. Minimal-window search ($MWS(d-r) \geq \beta - \delta$):

   This scheme is much like the one above, except instead of using quiescence search to estimate the merit of the children, a minimal-window search to a closer horizon at distance $d-r$ is used.

To establish how well the number of promising moves, as judged by each of the above schemes, correlates to an expected cut-node being a True-cut-node or not, the program was instructed to gather statistics about cut-nodes. When the program visits an expected cut-node it calculates the number of promising move alternatives in the position according to each of the above schemes. Then, after searching the node to a full depth to determine if it really is a cut-node, information about the number of promising moves is logged to a file along with a flag indicating whether the node is a True-cut-node.

The program gathered statistics on 100,000 nodes. The resulting data was classified into two categories, one with True-cut-nodes (expected cut-nodes that became cut-nodes), and the other with the False-cut-nodes (expected cut-nodes that became all-nodes). Approximately 2.5% of the expected cut-nodes fell into the latter category (i.e. were False-cut-nodes). Table 5.1 gives a summary statistic, contrasting the two categories. Each row in the table shows, for each category, the average number of promising moves ($\overline{x}$) as judged by each classification scheme. The standard deviation (σ) is also provided. We are looking for the scheme that best separates the True- and False-cut-nodes, that is, where the averages are far apart and the standard deviation is low. Thus, by comparing the averages and the standard deviations for the two categories

Table 5.1: Comparison of different schemes for identifying False-cut-nodes.

Method	True-cut-nodes		False-cut-nodes	
	$\overline{x}$	σ	$\overline{x}$	σ
NM	35.60	11.74	24.83	14.46
$HH > 0$	22.27	8.87	16.35	9.77
$HH > 100$	9.15	5.72	7.13	5.33
$QS() > \beta$	20.48	15.03	0.32	1.44
$QS() > \beta$-25	23.70	14.08	1.66	4.20
MWS(d-2) $> \beta$	20.62	14.88	0.17	0.55
MWS(d-2) $> \beta$-25	23.75	14.00	1.46	3.75

we can determine the scheme that best predicts which expected cut-nodes are False-cut-nodes.

In Table 5.1, it is interesting to note that even a simplistic scheme like looking at the number of legal moves shows a difference in the averages. However, the difference is relatively small and the standard deviation is high. The history heuristic schemes have lower standard deviation, but unfortunately the averages are too similar. This renders them useless. The methods that rely on search, $QS()$ and $MWS()$, do much better, especially those where δ (the β-cutoff margin) is set to zero.[1] Not only are the averages for the two groups far apart, but the standard deviation is also very low. From the data in Table 5.1 these two schemes look almost equally effective.

Therefore, to discriminate between them further, we filtered the data for the False-cut-nodes looking only at non-zero data points (that is, we only consider data points where at least one promising move alternative is found by either scheme). The result using the filtered data is given in Table 5.2. Now we can see more clearly that the minimal-window (MWS) scheme is a better predictor of False-cut-nodes. Not only does it show fewer false promises on average, but the standard deviation is also much lower. This means that

[1] In THE TURK, a δ value of 25 is equivalent to a quarter of a pawn.

Table 5.2: Comparison of selected schemes using filtered data.

Method	False-cut-nodes	
	$\overline{x}$	σ
$QS() > \beta$	2.31	3.20
$MWS(d-2) > \beta$	1.45	0.86

it infrequently shows False-cut-nodes as having more than several promising move alternatives. Even in the worst case there never were more than 6 moves listed as promising, whereas for the $QS()$ scheme at least one position had 32 wrong indicators.

The above experiments clearly support the hypothesis that there is a way to discriminate between nodes that are likely to become true cut-nodes and those that are not. As a result, we selected the shallow minimal-window searches (MWS) as the scheme for finding promising moves in multi-cut $\alpha\beta$-pruning.

5.4.2 Multi-Cut Parameters

Experiments were performed with different instantiations of the multi-cut parameters, not only to provide a better insight into how they alter the search behavior, but also to find the most appropriate parameter setting for the program. The program was tested against a suite of over one thousand tactical chess problems [66]. For each run different multi-cut parameter settings were used, and information was collected about both the total number of nodes explored, and the number of problems solved. The program was instructed to search to a nominal depth of 7-ply, and use normal search extensions and null-move search reductions. Basically, we are looking for the parameters that give the most node reductions, while still solving the same number of problems as the original program does.

Figure 5.2 shows the search effort under a range of parameter settings. The search effort is given as a percent of nodes searched by the standard

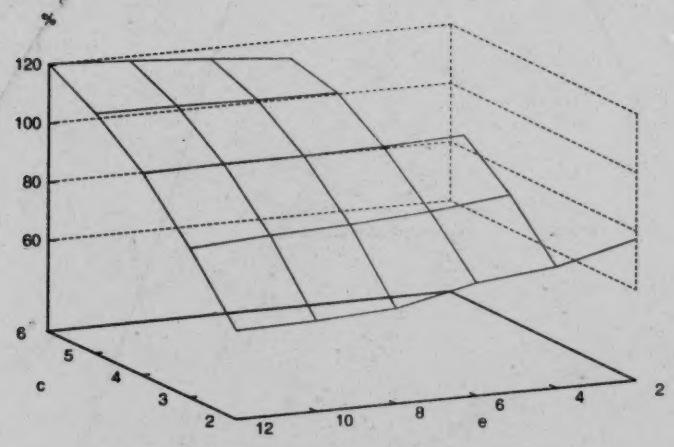


Figure 5.2: Search efficiency when $r = 2$.

version of the program. The depth reduction was fixed at 2, but the c and e parameters were allowed to vary from $2 - 6$ and $2 - 12$, respectively. We also experimented with different depth reduction factors, but we found that a value of $r = 1$ offers only limited node savings, while values of $r > 2$ were too error prone (see Table 5.3). As expected, the fewest nodes are examined for small values of c. For example, the program with $c = 2$ and $e = 12$ searches over 40% fewer nodes than the original program. However, the node savings decrease rapidly as c increases, breaking roughly even at $c = 4$, and searching considerately more nodes for higher values. We also see how e influences the search, although these changes are more subtle. An interesting observation is that for low values of c the total number of nodes decreases as e increases, but the opposite is true for higher values of c. This can be explained by the counter-acting effects we discussed earlier. For low values of c, we observe more *mc*-prunings as e increases, and the extra cutoffs more than offset the additional search overhead of each *mc*-prune search. However, for larger values of c there are far fewer additional cutoffs, and the increased cost of each *mc*-prune search starts to show. From looking only at this graph, one can deduce

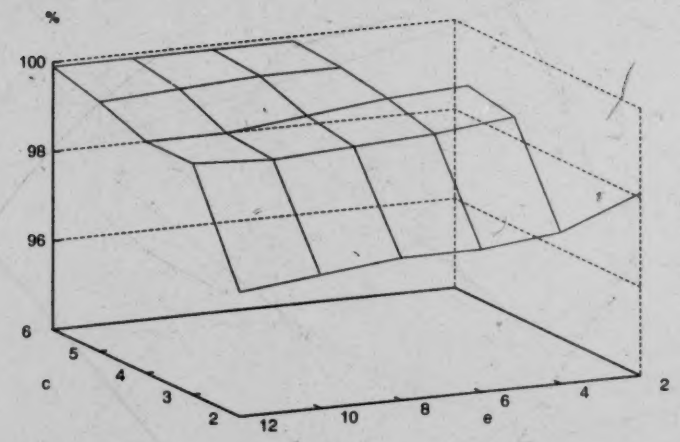


Figure 5.3: Decision quality when $r = 2$.

that a low value of c and a relatively high value for e results in the best search efficiency. However, we still have to look at the other side of the coin, namely the error rates associated with the different parameter settings.

Figure 5.3 shows a similar graph, except here we are looking at the percentage of problems solved (as compared to the standard version of the program). Most notable is the steep increase in the percentage of problems solved as c is increased from 2 to 3. However, increasing c further only yields slow improvement. There is also a slight trend towards improved accuracy as e is decreased, at least for the smaller values of c. This is understandable, since by decreasing e the criterion for *mc*-prune is being set more conservatively.

From the above data, setting $c = 3$ and e somewhere in the high range of $8-12$ looks the most promising. These settings give a substantial node savings (about 20%), while still solving over 99% of the problems that the standard version does. The data charting the two above graphs is provided in Table 5.3.

Table 5.3: $T_{mc(c,e,r)}$ searches showing the performance of different multi-cut parameter settings relative to a standard search, in both terms of % of nodes searched (Nod%) and problems solved (Sol%).

r	c	e	Nod%	Sol%	r	c	e	Nod%	Sol%	r	c	e	Nod%	Sol%
1	2	2	92.05	98.10	2	2	2	77.28	98.10	3	2	2	79.21	96.80
1	2	4	93.33	97.60	2	2	4	70.48	97.40	3	2	4	71.60	95.80
1	2	6	93.02	97.20	2	2	6	67.61	97.20	3	2	6	67.71	95.80
1	2	8	91.71	97.20	2	2	8	61.56	97.20	3	2	8	63.17	95.50
1	2	10	92.10	96.80	2	2	10	60.04	97.00	3	2	10	60.57	95.20
1	2	12	93.39	96.80	2	2	12	59.38	96.80	3	2	12	57.13	95.10
1	3	4	134.17	99.20	2	3	4	87.46	99.50	3	3	4	86.07	97.70
1	3	6	144.14	99.20	2	3	6	84.41	99.30	3	3	6	82.92	97.50
1	3	8	150.31	98.90	2	3	8	82.60	99.20	3	3	8	79.30	97.50
1	3	10	153.00	98.70	2	3	10	81.66	99.10	3	3	10	75.86	97.10
1	3	12	157.34	98.50	2	3	12	79.95	99.20	3	3	12	72.21	97.00
1	4	4	175.38	99.40	2	4	4	100.14	99.70	3	4	4	98.33	98.60
1	4	6	194.19	99.40	2	4	6	98.86	99.60	3	4	6	94.20	97.90
1	4	8	210.41	99.30	2	4	8	98.50	99.40	3	4	8	89.96	97.90
1	4	10	222.67	99.10	2	4	10	98.51	99.20	3	4	10	87.39	97.70
1	4	12	234.33	99.00	2	4	12	98.04	99.20	3	4	12	84.89	97.60
1	5	6	227.73	99.50	2	5	6	109.63	99.80	3	5	6	97.23	98.50
1	5	8	252.26	99.60	2	5	8	109.93	99.80	3	5	8	94.95	98.10
1	5	10	276.16	99.50	2	5	10	110.67	99.70	3	5	10	92.02	97.90
1	5	12	286.82	99.40	2	5	12	110.88	99.60	3	5	12	90.24	97.80
1	6	6	239.81	99.70	2	6	6	113.77	99.90	3	6	6	100.97	99.20
1	6	8	269.33	99.70	2	6	8	116.40	99.90	3	6	8	99.42	98.30
1	6	10	312.24	99.70	2	6	10	118.61	99.90	3	6	10	100.24	98.30
1	6	12	335.51	99.70	2	6	12	120.23	99.90	3	6	12	95.66	98.00

5.4.3 Multi-Cut in Practice

Ultimately, we want to show that game-playing programs using the new pruning method can achieve increased playing strength. Although the aforementioned experiments are useful in giving insight into the feasibility of the idea and the behavior of the search, they do not tell how beneficial the new method is in practice. For that actual games are needed. Generally, when using a speculative-pruning scheme, playing games is the only way to show the appropriate balance between improved search efficiency and added risk of overlooking good continuations. We experimented with multi-cut in two different game-playing programs, a chess program and a Lines-of-Action program.

Table 5.4: 80 game multi-cut chess match results.

$T_{mc(3,10,2)}$ versus T		
Time control	Score	Winning %
40 moves in 5 minutes	46 - 34	57.5
40 moves in 15 minutes	42 - 38	52.5
40 moves in 25 minutes	43.5 - 36.5	54.4
40 moves in 60 minutes	43 - 37	53.8
In total 320 games:	174.5 - 145.5	54.5

Chess

Two versions of the THE TURK were matched against each other, one using multi-cut pruning and the other not. Four matches, with 80 games each, were played using different time controls. To prevent the programs from playing the same game over and over, forty well-known opening positions were used as a starting point (see Appendix D.5). The programs played each opening once from the white side and once as black. Table 5.4 shows the match results. T represents the unmodified version of the program and $T_{mc(c,e,r)}$ the version with multi-cut implemented. We experimented with the case $e = 10$, $r = 2$, and $c = 3$ (i.e. 10 moves searched with a depth reduction of 2-ply and with 3 cutoffs required to achieve the *mc*-prune condition).

The multi-cut version shows a definite improvement over the unmodified version, scoring overall 175.5 vs. 145.5 points. In tournament play this winning percentage would result in about 35 points difference in the players' performance rating. Although this single set of experiments doesn't allow us to quantify the exact strength difference between the two versions (for that far more games are needed, preferably against many different opponents), we can state with over 90% statistical significance that the multi-cut version is the stronger.

Finally, as an additional insight, THE TURK gathered statistics about the

behavior of the multi-cut pruning. The search spends about 25%-30% of its time (in terms of nodes visited) in shallow multi-cut searches, and an *mc*-prune occurs in about 45%-50% of its attempts. Obviously, the tree expanded using multi-cut pruning differs significantly from the tree visited when it is not used.

Lines of Action

One of the strengths of the multi-cut method is that it does not use any game-specific knowledge (although the multi-cut parameters might need to be set differently depending on the game); thus, it is tempting to state that it can equally well be employed in other games. However, one has to be extremely careful with statements like that. Sometimes improvements that at first sight appear to be game independent, nonetheless turn out to be restricted to use in only one specific game. Often they depend on some hidden properties of the search space, present only in that particular game.

To see if multi-cut is beneficial in games other than chess, we implemented it in a Lines of Action (LOA) game-playing program. The program, YL, is the gold-medal winner from the 2000 and 2001 Computer Olympiad [13, 20], and has indisputably established itself as one of the strongest LOA player in the world (including both human and computer players). The game was invented by Claude Soucie in the 1960's, but has only recently attracted the attention of the AI community [81]. The game is played on an 8x8 checkers/chess board. Each player starts with 12 pieces on the board and the first player to connect all of them into one group wins. The rules of the game are described in the well-known book *A Gamut of Games* [67], but are also accessible on the World-Wide-Web.

As in the chess experiments, we matched two versions of the program against each other, one using multi-cut and the other not (but otherwise identical). The multi-cut parameters $c = 3$, $c = 3$, and $r = 2$ were chosen. The

Table 5.5: 622 game LOA match result.

$YL_{mc(3,3,2)}$ versus YL			
Program	Game results	Score	%
$YL_{mc(3,3,2)}$	+328 =16 -278	336 - 286	54.02
YL	+278 =16 -328	286 - 336	45.98

reason for setting $e = 3$ (as opposed to 10 in the chess experiment), is that the average branching factor in LOA is somewhat less than in chess; in a typical middle-game position there are generally between 20-30 legal move alternatives, whereas in chess the range is more like 30-40 moves. To prevent the programs from re-playing the same game over again, different opening positions were used as starting points. Unlike in chess, an established set of standard opening moves has not been established in LOA. This posed a problem when selecting which opening positions to use in our self-play experiments. To be objective we rejected the idea of pre-selecting the opening positions; instead we generated from the initial game state all possible game positions two moves into the game (one move for each side). This results in 311 different start positions. Because each program plays both sides of the opening, the self-play match consisted of 622 games in total. All games were played on Intel Pentium III computers using 30 minutes a game time controls. For each game the opposing programs played on the same computer, taking turns playing a move (the thinking-on-opponents-time feature was turned off to prevent the programs from competing for CPU time).

The outcome of the match is shown in Table 5.5. We see that the multi-cut version won 328 games, drew 16 games, and lost 278 games, resulting in a significant winning margin of 336 vs. 286 points. We also investigated how aggressively the multi-cut method prunes the game tree. For the same nominal search depth, the multi-cut program searches only roughly two-thirds

the nodes of the unmodified version. Thus, given the same amount of time, the program reaches deeper nominal search depths on average. The benefits of that clearly outweigh the effects of occasionally introducing a pruning error in the search.

One drawback of the approach used to select the opening lines is that we run the risk that many of the openings are lopsided, that is, directly from the opening one side establishes sufficient advantage to handily win the game. This would clearly favor the weaker program by making the outcome of the match look closer than it would otherwise be. We tried to estimate how profound this problem is in Lines of Action.[2] In Table 5.6 we view the games as 311 independent mini-matches, where each mini-match consists of the two games played from the same opening position. The multi-cut versions wins 69 of these mini-matches and loses 42. However, it is striking to see how many of them end in a draw, mainly because the same side (color) wins both the games. This could be an indication that a high number of the opening positions are unusually lopsided. If this really is the case we would need to find an independent and objective way to eliminate these openings. Nonetheless, despite the high number of drawn matches, the multi-cut version wins 64% more matches than the unmodified program. Assuming that the multi-cut version is the stronger, as the result clearly indicates, we would expect the winning percentage to be even higher if all lopsided positions were eliminated from the test suite.

The second advantage of looking at the data as mini matches is that it allows us to perform a standard statistical test to see if the performance improvement of the multi-cut version is statistically significant. If the unmodified version of the program played 311 mini-matches against itself, the expected

[2]In tournament checkers, opening positions are selected at random in a similar way. However, some of the openings have been removed (banned) because they lead to an easy win for one side.

Table 5.6: 311 mini-match LOA result.

$YL_{mc(3,3,2)}$ versus YL		
Program	Match results	%
$YL_{mc(3,3,2)}$	+69 =200 -42	+22.2 =64.3 -13.5
YL	+42 =200 -69	+13.5 =64.3 -22.2

outcome would be that they all end in a draw because the second game would be an exact replica of the first.[3] Thus, we can use a *student-t* statistical test to compare the mean of the win-draw-loss distribution of the multi-cut program to the expected distribution of a YL vs. YL match (all draws). We performed the test, and based on that we can state with over 99% statistical significance (t=2.5861, p=0.0099) that the mean of the former distribution is higher. This demonstrates with a high confidence that the multi-cut version is definitely stronger.

5.5 Related Work

The idea of exploring additional moves at cut-nodes is not entirely new. There exist at least two other variants of the $\alpha\beta$ algorithm that explore more than one alternative at cut-nodes, although the resulting information is used differently in our work.

The *Singular Extensions* algorithm [3] extends "singular" moves more deeply than others.[4] A move is defined as singular if its evaluation is higher than all its siblings by some specified margin, called the singular margin. Moves that fail high, i.e. cause a cutoff, automatically become candidates for being singular (the algorithm also checks for singular moves at pv-nodes). To determine if a candidate move that fails-high really is singular, all its siblings are ex-

[3]Because the program used real time-controls, there is a slight possibility that in exceptional circumstances some games would not be repeated, resulting in an occasional mini-match to end in a win for either side.

[4]The algorithm is also briefly explained in Chapter 2.

plored to a reduced depth. The move is declared singular only if the value of all the alternatives is significantly lower (as defined by the singular margin) than the value of the principal variation. Singular moves are "remembered" and extended one additional ply on subsequent iterations. One might think of multi-cut as the complement of singular-extensions: instead of extending lines where there is seemingly only one good move, it prunes lines where many promising (refutation) moves are available.

The *Alpha-Beta-Conspiracy* algorithm [54] is essentially an $\alpha\beta$-search that uses conspiracy depth, instead of classical ply depth, to decide when to stop searching a branch. The conspiracy depth is updated at each node in the tree, but instead of reducing the depth always by one ply, it can be reduced by a fraction of a ply, all depending on how many good alternative moves there are. The fewer alternatives, the smaller will be the conspiracy depth reduction. Quiescence searches are used to establish the number of good alternative moves. This algorithm encourages forced lines to be searched more deeply. Another distinct feature of the algorithm is that two separate conspiracy depth parameters are used, one for each player. At each level, only the conspiracy depth parameter for the player to move is updated. The search explores a branch until either both conspiracy depths parameters converge to zero, or alternatively, when the conspiracy depth for the player to move is zero and a static evaluation delivers a cutoff. However, empirical results using this algorithm are not favorable.

5.6 Conclusions

We have shown that there is a strong correlation between the number of promising move alternatives available at an expected cut-node, and the node becoming a True-cut-node. We introduced a new pruning method, multi-cut, that exploits this correlation. Furthermore, to show the promise of the idea,

we implemented and experimented with the technique in two different game-playing programs. Our experimental results give rise to optimism. In match play using two different games as a test bed, the versions using the new pruning method consistently outplayed the unmodified program versions. Our new search method, while expanding a tree that is radically different from that of the $\alpha\beta$ algorithm, significantly improved the playing strength of the two game programs we used for our experiments.

The multi-cut method is still relatively new, and has likely not yet matured to the state of achieving its full potential. For that, much more tuning and testing is needed (years of experience has shown that it usually takes a long time for a new search enhancements to fully evolve and its potential to be fully realized). There is definitely scope for improvement through further tuning and enhancement. For example, one promising avenue for improvement is to parameterize the pruning using variables instead of constants for c, e, and r; that way their values can be adjusted dynamically as the game/search progresses. Another possibility, to minimize the risk even further, is to use a *layered multi-cut*; that is, require that a multi-cut occurs on a least two different places along the search path before pruning takes place. With programs searching deeper and deeper every year, that approach starts looking more feasible. Also, the multi-cut method does not utilize any game-specific knowledge, we deliberately made this decision to make the approach as domain independent as possible. This is both a strength and a weakness. It is quite possible that the performance of the method can be further enhanced by looking at domain-specific properties.

Our experiments show that there is room for innovative domain-independent pruning methods, based on exploiting the structure of a critical tree. The multi-cut method as described and implemented here — although promising — is not the only way of using the information about the number of good

move alternatives at cut-nodes, and by no means necessarily the best. The multi-cut method, to the best of our knowledge, has been successfully adopted by some of the world's strongest commercial chess programs.

Chapter 6

Learning Search Control

Tactics is knowing what to do when there is something to do; strategy is knowing what to do when there is nothing to do.
--Savielly Tartakower

In the previous chapters we investigated methods for reducing the size of the search tree. The rationale behind these methods is that by exploring less deeply continuations that look futile, more time may be invested in exploring interesting lines more deeply. In this chapter we look at search extensions, an alternative way of adding selectivity to depth-first search.

6.1 Introduction

In the domain of planning and scheduling, machine learning methods have been applied successfully to improve both the search efficiency [57, 55] and, more recently, the quality of the produced plans [62, 31]. These methods work primarily by deriving and refining control rules. Unfortunately, such a rule-based approach is not feasible for learning search control in two-person games such as chess, checkers, and Othello. First of all, many decades of experience have proven that it is difficult to produce search-control rules that generalize well from one game position to the next. Secondly, in competitive play the game-playing programs must meet external time constraints, often having only

a few minutes to decide on a move. Efficiency is therefore of paramount importance and the overhead of manipulating complex search-control rules can easily outweigh the possible benefits.[1] This calls for a different approach for learning search control.

In view of the above, it is not surprising that machine learning in games has not focused so much on the search as on other aspects of the game, where existing learning techniques are more readily applicable. For example, many different schemes exist for learning evaluation function parameters [68, 69, 9, 25, 4], and more recently, work is being done on dynamic adjustment of opening-books [26, 42]. However, attempts to improve the search efficiency in two-person games have not been particularly successful. For example, *explanation-based learning* [57] and *case-based reasoning* [45] approaches, although interesting, have yet to demonstrate an overall improvement in the search efficiency. Also, early attempts to use patterns to guide the search were only moderately successful [80]. In games like Go, where search is of a lesser importance, some success has been achieved recently by learning search-control rules [28]. For an overview of methods for learning in games see, for example, Fürnkranz [34, 35].

In addition to the above approaches, some of the search enhancements already widely employed in adversary search may be viewed as simple forms of learning. Two such enhancements are the *history-heuristic* [71] and *permanent hash-table entries* [73]. The former enhancement keeps a table storing a merit value for each move indicating how well it has done so far. This information is used to improve move ordering during the search. The latter method is a simple form of rote learning: the program simply keeps track of losing positions from previous games, such that it can avoid repeatedly losing games in the same

[1] For example, the top competitive chess programs typically explore close to a million chess positions per second on contemporary PCs.

fashion. More recently there have been attempts to improve move-ordering schemes for several games using specifically trained neural networks [38, 47].

In this chapter we introduce a novel method for learning search control in two-person games. The method is equally suited to learn either during online play, or by analyzing game positions offline. Before we describe the learning system it is worthwhile to briefly review search-control strategies in two-person games.

6.2 Search Control

As previously discussed, $\alpha\beta$-based algorithms are almost universally employed by game-playing programs in board games such as chess, checkers, and Othello. The search efficiency of the algorithm can be improved in a couple of ways: either by improving the move ordering (expand fewer nodes of the game tree) or, alternatively, by being more selective of how deeply to explore each line of play (can find a good solution in an early iteration). We previously mentioned a few move-ordering strategies, but here we are only concerned with selectively deciding which nodes to search deeper or shallower.

6.2.1 Search Extensions

The number of nodes visited by the $\alpha\beta$ algorithm grows exponentially with the search depth. The question now becomes: how can a program best use the available time to find a good move? Although the basic formulation of the $\alpha\beta$ algorithm explores all continuations the same number of plies, it has long been evident that this is not the best search strategy. Ideally, interesting continuations are explored more deeply while less interesting alternatives are terminated prematurely. In chess, for example, it is common to resolve forced situations, such as checks and recaptures, by searching them more deeply. The search efficiency — and consequently the move-decision quality — of the $\alpha\beta$

algorithm is greatly influenced by the choices of which lines are investigated deeply and which are not. Therefore, the design of a search-extension scheme is fundamental to any game-playing program using an $\alpha\beta$-like algorithm. Several studies have been conducted to quantify the relative importance of various extension schemes [3, 82, 8]. Unfortunately, the more elaborate the search-extension scheme, the more difficult it is to parameterize to achieve its full search-efficiency potential. In here we introduce a method for automatically tuning these parameters.

6.2.2 A Unified View

Although they are all based on the same principles, the specific search-extension schemes employed by the various game-playing programs differ somewhat from one program to the next. Therefore, to make our learning system as widely applicable as possible, we introduce a unified framework that attempts to encapsulate the various implementations.

Figure 6.1 shows an example game tree that is being searched to an arbitrary depth, say d'. For any node x in a tree, let P_x stand for the move path leading from the root of the tree to that node. For example, the path P_H consists of the move sequence connecting nodes *A-B-E-F-G-H*. In our unified framework, a function $D(P, \vec{w})$ decides how far to expand each line of play. The current move path is expanded until:

$$D(P_x, \vec{w}) \geq d'.$$

The function takes the current move path as its first argument and returns its *depth*. Note that the depth of the path is not necessarily the same as its length. The *length* is simply the number of moves on the path, whereas the *depth* can be determined by whatever criteria we like. When a path's depth is less than its length, the path will be extended beyond the nominal search

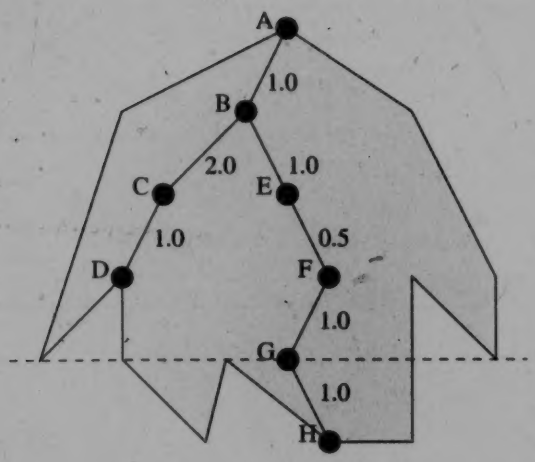


Figure 6.1: Search-extension schemes - a unified view

horizon. Search reductions occur when the opposite is true. The special case where the depth of each move path equals its length results in a search strategy that explores all continuations the same fixed number of plies. The second argument of D, $\vec{w}$, is a vector of search-control parameters that influence the depth calculations. These parameters are made explicit because they are the ones we want to learn. In practice, there are probably additional parameters that must be passed to the function, such as the root game position and the α and β search bounds. However, to simplify our notation we do not show them, but we may assume arbitrary many such parameters (as long as they are not a function of $\vec{w}$). The only restriction we put on the depth function is that it is *monotonically non-decreasing*, that is, the depth of a move path will never decrease by adding more moves to the end of the path.

Our framework is quite general and incorporates most of the different search extension schemes known to us. On the other hand, when implementing a competitive game-playing program, one typically does not have an explicit function for calculating the depth of the move path at each frontier node — instead the depth is updated incrementally. However, this does not pose a

problem as long as there exists a conceptually equivalent formulation in the form of a depth function.

6.2.3 Fractional-Ply Extensions

Now we show how a commonly used search-extension scheme, often referred to as *fractional-ply extensions* [49, 41], can be trivially formulated within the unified framework (the game-playing program we use for our experiments employs this type of extension scheme). The existence of predefined move classes is assumed, where each class has a weight associated with it. Examples of move classes in chess could be checking moves, recaptures, and advanced passed-pawn pushes. During the search, each move is categorized as belonging to one of the move classes, and the depth of the current move path is the sum of the class weights of the moves on the path. Referring back to Figure 6.1, the numbers on the paths show the class weight of each move. For example, the depth of path P_D is $1.0 + 2.0 + 1.0 = 4.0$, and similarly the depth of P_H is 4.5. More formally, if we assume that there are N predefined classes, the depth function becomes:

$$D(P, \vec{w}) = \sum_{j=1}^{length(P)} w_i \mid i \equiv Class(m_j)$$

where m_j is the j-th move on the path, the vector $\vec{w}$ contains the weights for each of the N move classes (the element w_i is the weight of class number i). The $Class(m_i)$ function categorizes each move as belonging to one of the move classes $1, ..., N$. The search-control parameters to be tuned are the weights of the move classes.

6.3 The Learning System

The main advantage of the general framework above is that search-control learning can now be viewed as a function-approximation task, namely approx-

imating the $D(P, \vec{w})$ function. In other words, the task of the learning system is to find the most appropriate weight vector $\vec{w}$. As with all such tasks, we must decide on the training experience, the exact representation of the target function (i.e. the function we are trying to approximate), and the algorithm for adjusting the weights.

6.3.1 Training Experience

We want the game-playing program to learn from its mistakes and adapt the search behavior accordingly. However, for that to be possible the program must first recognize when it makes a mistake. For human players this is generally not that difficult a task. Experienced players will identify where they went wrong in a post-mortem analysis of a game; the player might have over-estimated his or her chances in a particular position, may have chosen a dubious plan, or simply overlooked some tactical continuations. On the other hand, identifying mistakes is a challenging task for a computer player. Obviously, if the program loses a game in an abrupt fashion it is clear that a mistake was made, but to pin-point exactly what move or moves were the cause of the defeat is not trivial. This problem is sometimes referred to as the *credit assignment* problem, and it's hard in the general case. However, there are situations where mistakes can be identified with a high degree of certainty.

Figure 6.2 shows a search tree for a game in progress: the moves connected by the solid lines have already been played, and currently the program is searching game position C. Based on the search the program determines the principal continuation to be $m_1,...,m_n$ (shown as dotted lines), and assesses the position as having a value v_C. Now, assume that when it was the program's turn to move at position A the assessment was significantly higher, or

$$v_C < v_A - \tau$$

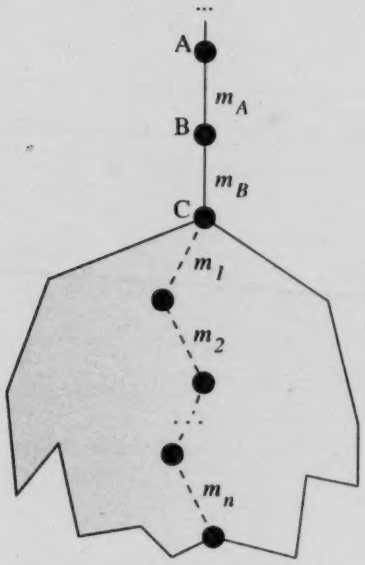


Figure 6.2: Identifying mistakes.

where τ is a positive constant representing the significance margin. The program now evaluates its chances much poorer than just a move ago; clearly something must have gone wrong! But what caused this undesirable change of fortune? One of two things could be responsible. It might be that position A was already bad but that the program just didn't realize it. Alternatively, it could be that position A was fine and the move m_A was a mistake — and only now does the program see the bad consequences of that move. However, in either case, position A was assessed incorrectly. Thus, A is referred to as a *critical position*, and the move sequence m_A, m_B, m_1, ...,m_n is known as the *solution path* of the position. The basic assumption that we make here is that *if the search is to correctly assess position A, its solution path (S_A) must be fully explored.*[2] This implies that the game tree for position A needs to be explored to the depth of its solution path. Critical positions and their solution

[2]Note, this is not a sufficient condition for correctly assessing the position, because other lines in the game tree might also need to be explored more deeply. We are only assuming this to be a necessary condition.

paths form the training input for our learning system. Many existing problem test suites consist of a collection of game positions and their corresponding solution paths, meaning that they can also serve as a training input for our learning method.

It is interesting to note that it is not instructive to learn from cases where the positional estimate increases from position A to C. The reason is that the in-between move made by the opponent, that is move m_B, might simply be a blunder. The search might have explored that move at position A deeply enough to correctly discard it as a bad move, in which case there is no need to change the search parameters.

6.3.2 Target Function

The function we want to approximate is the depth function $D(P, \vec{w})$. However, it is not clear how the training experience from the above example helps us do exactly that. Although we know position A and its solution path, there is no information about the "correct" depth for the path. This renders supervised learning methods practically useless. Instead, we must go about this indirectly. One way of reformulating the problem is to ask: which weight vector results in the search expanding the fewest nodes possible to find the given solution path? Given our previous assumption, we know that to find the solution the position must be expanded to at least the depth of its solution path. Therefore, we alter the question slightly to: *when expanding the position to the depth of its solution path, which weight vector causes the search to expand the fewest nodes possible?* Answering this question is not trivial since changing the weights affects not only the solution path but also other parts of the search. Without actually performing the search we have no way of telling how many nodes it will take to explore position A that deeply (and during a game we cannot revisit the position to search it again).

Suppose that we have a cost model

$$C(p, \vec{w}, d)$$

that predicts how many nodes it takes to search position p to depth d using weight vector $\vec{w}$. We could use this cost model to answer the question we posed above, that is, predicting the number of nodes it takes to expand position A to the depth of its solution path as:

$$C(A, \vec{w}, D(S_A, \vec{w})).$$

More generally, given a set of training samples, T, where each sample is a pair $< p_t, S_t >$ consisting of a game position (p_t) and a solution path (S_t), we are interested in finding the weight vector $\vec{w}$ that minimizes the total number of nodes (as estimated by the cost model) that it takes to "solve" all the samples. In other words we want to minimize the function:

$$F(\vec{w}) = \sum_{t \in T} C(p_t, \vec{w}, D(S_t, \vec{w})).$$

If the function $C(p, \vec{w}, d)$ were known this could be done numerically, or even analytically. However, for games of any complexity it is practically impossible to analytically model this function. Not only does it depend on the weight vector but also on various positional features. A key observation here is that it is *not necessary to formally model the function over the entire search space* to be able to minimize it. When using a hill-climbing-like method, it is sufficient to be able to approximate it for any individual point in the search space. Fortunately, we have a way of doing that, as we show in section Section 6.4, but let us first concentrate on the learning algorithm.

6.3.3 Learning Algorithm

A standard hill-climbing method, *gradient-descent* [56], is used to minimize $F(\vec{w})$. Although the method guarantees finding a global minimum only for

concave functions, nonetheless, in practice it is a highly effective heuristic approach to optimization and forms the basis of various learning systems (e.g. the back-propagation rule in artificial neural networks). The gradient-descent method starts with some initial setting for the weight vector $\vec{w}$, and then repeatedly iterates over all the training samples, each time updating the weight vector in the *opposite* direction of the gradient. The gradient of $F(\vec{w})$ specifies the direction of weight changes that produce the steepest increase in the value of $F(\vec{w})$. Therefore, by adjusting the parameters in the opposite direction, one expects the value of the function to incrementally decrease at each iteration. This process is continued until some termination condition is met. The condition could be as simple as doing a fixed number of iterations, or a more elaborate one like: continue until the progress becomes negligible. The gradient provides the sign and relative size of each weight change, while the step size — that is, how much the weights are altered — is controlled by the learning rate μ. The learning rate is typically decreased after each iteration to avoid stepping over the minimum and to ensure eventual convergence. The exact procedure for decreasing μ depends on the search domain, and is often determined by trial and error.

Our implementation is outlined as Algorithm 12 below. The algorithm starts by initializing the search control parameters (w_i) to 1 (lines 2-4), and then repeatedly iterates over the test suite data T (consisting of game positions p_t and corresponding solution paths S_t). In our experiments, for simplicity, a fixed number of iterations is done. Before starting each iteration we initialize the variables that record the total node-count information (lines 7-10). The variable *nodes* indicates the total number of nodes that our cost model predicts it will take to solve all the problems in the test suite, whereas each $\Delta nodes_i$ stores how much this node count would change if we were to alter the corresponding search control parameters, w_i. The node-count information

Algorithm 12 *LSC*

1: // Initialize $\vec{w}$
2: **for** $i = 1, N$ **do**
3: $w_i \leftarrow 1$
4: **end for**
5: // Iterate until a sufficiently good $\vec{w}$ is found.
6: **while** *not terminate* **do**
7: $nodes \leftarrow 0$
8: **for** $i = 1, N$ **do**
9: $\Delta nodes_i \leftarrow 0$
10: **end for**
11: **for all** $(p_t, S_t) \in T$ **do**
12: $nodes = nodes + C(p_t, \vec{w}, D(S_t, \vec{w}))$
13: **for** $i = 1, N$ **do**
14: $\Delta nodes_i \leftarrow \Delta nodes_i + \partial C(p_t, \vec{w}, D(S_t, \vec{w}))/\partial w_i$
15: **end for**
16: **end for**
17: **for** $i = 1, N$ **do**
18: $w_i \leftarrow w_i - \mu \ \Delta w_{max} \ (\Delta nodes_i / nodes)$
19: **end for**
20: $\mu \leftarrow Decrease(\mu)$
21: **end while**

accumulates as we go through the test suite sample by sample (lines 11-16). The gradient (line 14) is used to tell how much the node count will change if a weight were to be altered. After finishing looking at all the game positions the search control parameters are updated proportionally to how much a change in them will affect the total node count (lines 17-19). The Δw_{max} constant is used for controlling the step size. Basically, a parameter change that causes 100% increase in the node count would result in a weight change of exactly Δw_{max} (given a learning rate of 1.0). Larger or smaller node count changes are adjusted proportionally. Finally, before starting the next iteration, the learning rate (μ parameter) is decreased.

A detail one might have noticed is that there is no direct reference to the actual game-playing program in the learning algorithm, only to the cost model. How can that be? The answer is that our cost model uses the game-playing

program to provide information about how many nodes the search actually expands.

Although we show the algorithm here as operating on an existing test suite of training samples, it is also suitable for learning from online game play. Then, instead of updating the weight vector after each iteration, it is updated after each training sample (or a subset of samples). This is a more convenient approach when learning during online game play, since we want to update the weights either immediately after encountering a critical position (see Section 6.3.1) or, alternatively, between games. This approach is sometimes referred to as *incremental gradient-descent* [56]. When using an incremental version of the algorithm it is important to use a slower learning rate (a smaller μ) to make sure the weights are not changed drastically based only on a single learning sample. An alternative approach would be to copy to a log file all critical positions and solution paths encountered during online play, and then learn offline from the resulting test suite using Algorithm 12.

6.4 Modeling the Search

So far we have assumed the existence of a cost model, but at the same time implied that it is impossible to accurately model the search. This seems paradoxical. However, as we mentioned before, it is not necessary to formally model the search over the entire search space. When traversing the hypothesis space of possible weight vectors, the gradient-descent algorithm requires information about only a few individual points in the search space. Fortunately, we have a way of approximating these points by using actual searches!

6.4.1 Cost Model

The cost model assumed by the learning algorithm returns an estimate of how many nodes the search expands when position p is explored to depth d.[3] Because the node count typically grows exponentially with increased search depth, the cost function must be of the basic form:

$$C(p, \vec{w}, d) = B(p, \vec{w})^d. \tag{6.1}$$

The $B(p, \vec{w})$ function measures the growth rate of the search. For example, $B(p, \vec{w}) = 4$ means that it takes 4 times as many nodes to search position p to depth $d + 1$ than to depth d. Even though the game trees themselves are highly irregular, the model above can be used as long as the growth rate is almost constant with respect to the search depth.

It is important to understand how altering the search-control parameters $\vec{w}$ affects the node-count estimate. Recall that for any given position p and corresponding solution path S, we are interested in knowing

$$C(p, \vec{w}, D(S, \vec{w})).$$

Modifying any weight has two fundamental effects:

- the exponential growth rate $B(p, \vec{w})$ changes, and
- the required search depth $D(S, \vec{w})$ is affected.

Typically, these two are counter-acting, for example, a change that reduces the depth of the solution path also tends to inflate the growth rate of the search. Intuitively, one would expect that altering the weights such that the required search depth is reduced would result in the smallest node count. However, it is

[3] Alternatively, one could measure the running time of the algorithm. However, that measure is a little more problematic because of hardware dependence, and non-deterministic behavior when running experiments on a multi-user platform. In any case, the number of nodes explored per second by the search algorithm is fairly constant within each phase of the game, and so these two measures are approximately equivalent.

quite possible that the modified weights will affect the exponential growth of the search in such a way that the estimated node count for the reduced search depth will indeed be higher than the one before. The right balance needs to be found.

Algorithm 12 needs to know the partial derivatives of $C(p, \vec{w}, D(S, \vec{w}))$ (see Appendix C for details), or

$$\frac{\partial C(p, \vec{w}, D(S, \vec{w}))}{\partial w_i} = C(p, \vec{w}, D(S, \vec{w}))\left(\frac{D(S, \vec{w})}{B(p, \vec{w})}\frac{\partial B(p, \vec{w})}{\partial w_i} + \ln(B(p, \vec{w}))\frac{\partial D(S, \vec{w})}{\partial w_i}\right) \tag{6.2}$$

The depth function $D(S, \vec{w})$ is assumed to be known in our model, and so are its derivatives. For example, in the fractional-ply extension scheme we mentioned earlier, the derivatives are simply the number of moves on S that belong to the i-th move class). The only unknown quantities in the above equation are therefore the $B(p, \vec{w})$ function and its partial derivatives.

6.4.2 Approximating $B(p, \vec{w})$ and its Partial Derivatives

In our cost model the growth-rate function $B(p, \vec{w})$ is constant with respect to the search depth.[4] Therefore, by knowing the node count for only a single search depth we can determine the growth rate. For instance, in the example given in Figure 6.2 we know how deeply position A was actually explored, say to depth d_A, and how many nodes were expanded, say n_A. Presumably, d_A is less than the depth of the solution path of position A. Now, by substituting d_A and n_A in for d and $C(p, \vec{w}, d)$ in equation 6.1, respectively, we get:

$$n_A = B(p, \vec{w})^{d_A} \Rightarrow B(p, \vec{w}) = n_A^{\frac{1}{d_A}}. \tag{6.3}$$

[4]Note that this does not imply that the search trees need to be of a uniform width or height, only that the average number of nodes ratio between a $d+1$ and a d ply deep search is approximately constant. This is the case in most games we work on, albeit, there are minor fluctuations between even and odd search depths. This is not an issue but, if necessary, one could trivially adjust the estimates to account for this.

The resulting approximation of the growth rate will allow us to use the cost model, and we can now estimate how many nodes the search will explore when expanding position A to the depth of its solution path. Because, in practice, the growth rate is not truly constant, this in only an estimate. Nonetheless, given that the depth d_A is reasonably close to the depth of the solution path, the estimate will be sufficiently accurate (the experimental result provided in Chapter 7 further supports this claim).

The partial derivatives are more problematic. Recall that the partial derivatives simply state how much the value of the function is expected to change if each weight is increased by a small amount, they can be approximated as:

$$\frac{\partial B(p,\vec{w})}{\partial w_i} = \frac{(B(p,\vec{w}+\vec{\Delta}_i) - B(p,\vec{w}))}{\delta_i} \tag{6.4}$$

where $\vec{\Delta}_i$ is a vector whose the $i-th$ element equal to δ_i and all the other elements zero. This requires us, though, to know the value of each of the $B(p,\vec{w}_i+\vec{\Delta}_i)$. One approach to come up with these values is to perform N (number of weight parameters) additional searches using a differently altered weight vector each time, and then use equation (6.3) to estimate the growth rate of the search for each of the altered weight vectors. Unfortunately, this is not feasible because of our requirement that the learning system be used during online play. Instead, *during the normal search we simultaneously estimate for each of the N altered weight vectors how many nodes would be expanded if they were used.* In addition to the normal depth, separate depths and node counts are recorded for each of the modified weight vectors $(\vec{w}_i+\vec{\Delta}_i)$. The node-count information gathered this way allows us to estimate each of the $B(p,\vec{w}_i+\vec{\Delta}_i)$ in the same way as before using equation (6.3).

This process is illustrated in Figure 6.3 for the fractional-ply extension scheme (see Figure 6.1). Assume that the tree shown is expanded using weight vector $\vec{w} = \{1.0, 2.0, 0.5\}$, that is, there are three move classes with weights

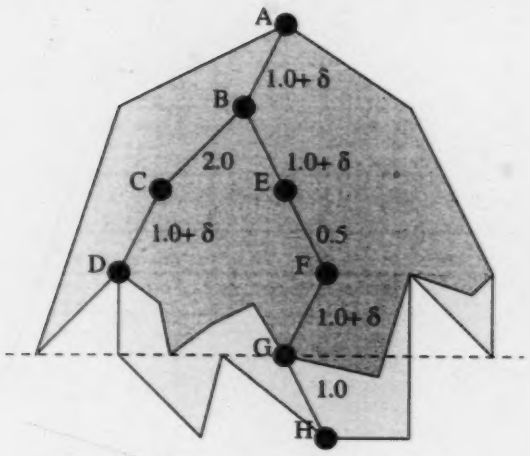


Figure 6.3: Approximating $B(p, \vec{w}_1 + \vec{\Delta}_1)$.

1.0, 2.0, and 0.5, respectively. The dark shaded area shows the subtree we would expect the search to expand when using an altered weight vector $\{1.0 + \delta_1, 2.0, 0.5\}$ $(\delta_1 > 0)$. At position G, for example, if the depth $D(p_g, \vec{w} + \vec{\Delta}_1)$ exceeds the search-depth limit, node H would not be expanded. Therefore it is not included in the total node count for that weight vector. The node-count information for the other two modified weight vectors is simultaneously gathered in the same way (not shown in the figure). A detailed example of this technique is given in Appendix B.

Note that this approach only approximates how many nodes are searched; if we really were to use a modified weight vector different values would be propagated up the tree, likely causing another set of branches to be expanded in some of the subtrees. Nonetheless, this approximation gives us a pretty good idea about the sensitivity of the search to changes in the search-control parameters. For this approximation to work, each of the weights must be altered such that the move paths become shorter — otherwise, the actual search would terminate before the altered depths reach the search-depth limit. In the previous example this means adding a positive constant to each of the

weights. However, because we do not put any restrictions on the form of the depth function, in other extension schemes this might imply that a weight must be reduced. One can even envision schemes where changing a parameter in either direction causes some move paths to shorten but others to lengthen. In such cases it might be possible to replace the troublesome parameter with two new ones, such that a parameter adjustment now causes consistent changes in the move path depths. When that is impossible, it might be necessary as a last resort to explore some paths in the tree beyond the depth the actual weight vector does, although, this would impose undesirable overhead on the search.

6.5 Experimental Results

To obtain practical experience with the learning method we implemented it in the chess program CRAFTY [41].[5] The program uses a fractional-ply based extension scheme with five different move categories: checks, re-captures, forced-replies to checks (i.e. only one legal reply), advanced passed pawn-pushes, and null-move threats. The last move-class, null-move threats, does not fit directly into the framework we introduced earlier. The reason is that a move can only be classified into this category by actually performing a null-move search. Thus, to keep things simple, we chose to disable it in our experiments. We ran two independent sets of experiments. In the first, the program was trained using an existing suite of chess-problems, while in the second the program learned during actual game play.

[5] CRAFTY is one of the strongest, if not the strongest, of the chess programs whose source code is publicly available. On the online chess servers it consistently ranks among the highest rated players, out-performing both some of the commercial chess programs and strong chess masters. The source code is publicly available via ftp at ftp.cis.uab.edu/pub/hyatt. Our learning scheme was originally implemented in version 16.4 and later re-implemented in versions 16.17. The results reported here are based on that latter version.

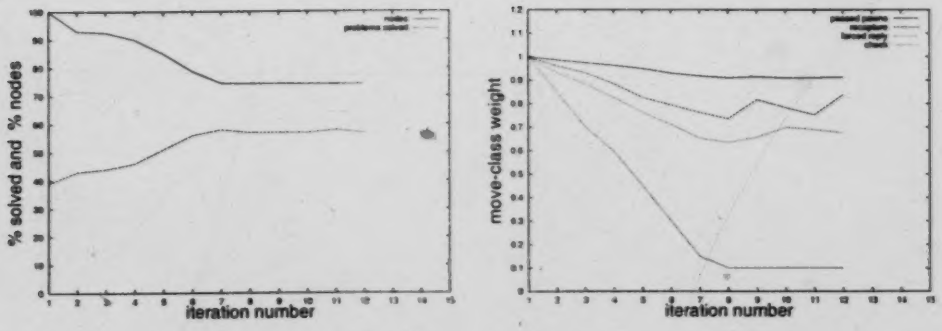


Figure 6.4: Learning results.

6.5.1 Test Suite

Within the computer-chess community it is common practice to benchmark the performance of chess programs against standard test suites. In the first set of experiments we observed the performance improvement of the program as it learned using the well-known ECM test suite [48]. This suite consists of 879 (mostly tactical) middle-game chess positions. Initially, the weights of the move categories were set to 1.0, and allowed to vary within the range [0.1,2.0]. For each of the problems, if the right move was not found after examining half a million nodes the search was stopped.

In Figure 6.4, the graph to the left shows the program's improvement from one iteration to the next. The dotted line shows the percentage of problems solved (out of the 879), and the solid line shows the total number of nodes searched relative to the first iteration (306 million nodes). The learning algorithm minimizes the total number of nodes required to solve the problems. The increase in problems solved follows indirectly as a side effect! The performance improves significantly as we can see. After only a few iterations the values have converged and the total node count is reduced to 74% (230M) of the original, at which level 57% (508) of the problems are solved correctly as opposed to 39% (346) in the beginning. The right-hand graph, on the other hand, shows

how the move class parameters evolve. The check extension weight rapidly drops to the minimum value, i.e. 0.1. This indicates that check moves are a particularly important category for extensions. The other weights also decrease, but more gradually, and finally converge to a value between 0.6 and 0.9 (more specifically the forced-replies, re-captures, and passed-pawn pushes classes have values of 0.68, 0.84, 0.91 respectively). The weight for the re-capture class is still oscillating around its optimal value. The reason is that we used a fixed learning rate, as opposed to decreasing it between iterations. Decreasing the learning rate can sometimes lead to a premature and false convergence of the parameters, something we wanted to avoid. On the other hand, by keeping the learning rate fixed one can experience oscillating behavior like this, where the method steps around the minimum without agreeing on one specific parameter value.

Many test suites, including the one we used, provide only the best-move for each position instead of the complete solution sequence. Because our learning method requires that the full solution-path be known, we had to make some compromises. If the best move returned by the program agrees with the move suggested by the test suite, we assume that the principal-variation given by the program represents the correct solution path. This path is stored and used in the current and all subsequent iterations. However, if the move returned does not agree with the test suite and there is no previously stored solution path for that particular problem, we simply ignore the problem. As a consequence, in each iteration we are minimizing the total number of nodes needed to solve only a subset of the problems in the test suite, that is, for those problems for which we have been able to derive a solution path. However, this subset gradually increases with each iteration and hopefully converges to a significant portion of the total test suite. A different approach that we could have taken is to find solution paths for the positions in the test suite, by pre-searching them

to a much greater depth. The drawback of that approach is that a few of the more difficult problems require extremely deep searches to be solved. These few problems would dominate the total node count needed to solve all the problems. This is undesirable, and the compromise approach we take avoids this problem altogether.

6.5.2 Game Playing

In the second set of experiments the program learned from playing games, instead of using a test suite of game positions. A version of the program using the learning scheme played 100 games against an unmodified version of the chess program (with a 5 minute time limit for each side for completion of an entire game). As before, the move class weights of the learner are initialized to 1.0. The program learns from critical positions encountered during the game. The threshold for a position to be considered critical is an evaluation drop of 1/3 of a pawn. Once the game position of the learner is considered to be lost (the position evaluation is more than 3 pawns down) the learning is disabled for the rest of the game. The reason is that once the position is already significantly worse, it is almost inevitable that one will lose more material and eventually the game. To learn from such losing examples is not particularly instructive.

The chess program used in the experiments distinguishes between three different game phases: the opening, the middle-game, and the end-game. The program evaluates game positions differently depending on which game phase it is in, but search extensions are done the same way in all phases. However, by automating the tuning-process of the weights we can easily learn a different set of weights for each game phase. Thus, our learning program is set up to use three different set of weights, one for each game phase. In our experiments, we did not receive any learning samples in the opening phase. This is not

Table 6.1: Learned weights.

Move class	Hand-set weights	Learned weights	
		Middle-game	End-game
Checks	0.00	0.30	0.10
Re-captures	0.25	0.10	0.10
Forced-reply	0.25	0.10	0.15
Passed-pawn push	0.25	0.47	0.25

surprising because the opening phase is typically rather short and the resulting positions are usually equal in value.

Table 6.1 shows the learned weights for middle game and end-game play, and how they compare to the weights used by the unmodified program (hand-tuned by the author of CRAFTY, a leading computer-chess expert). The learned weights differ substantially from the hand set ones. Also of interest is how the learned weights for the middle and endgame differ. We note that check extensions and passed-pawn pushes are extended more aggressively in the end-game, as one might hypothesize. To evaluate the quality of the learned weights, we matched six different versions of the program against each other. The only difference between the versions was the value of the search-control parameters. Each match consisted of 100 games played at time controls of five minutes per game.[6] To prevent the programs from repeating move sequences in the opening, each game was started from a different, well-established opening position (see Appendix D.6). The programs played each starting position once as White and once as Black.

The first program version, C_{online}, uses the learned weights shown in Table 6.1. For the opening-phase the same weights are used as for the middle-game. The C_{ECM} version uses the weights learned by using the ECM test

[6]The matches were played on Intel PII/400 and PIII/450 computers. Each match was played on a single computer. In the chess-program, all the default parameter settings were used, except that pondering (thinking on opponent's time) was turned off. Otherwise, the programs would compete for the computer's CPU time.

suite as training input, whereas the C_{hand} version uses the hand-set weights. The three remaining programs treat all the move-extension classes the same. In the C_{100} version all the weights are set to 1.00, which is the same as not using any search extensions. The C_{010} version extends aggressively on all move classes (all weights set to 0.10), whereas the C_{050} version uses more conservative extensions (all weights set to 0.50). The result of the matches is shown in Table 6.2. The program using the parameters learned from game play performs the best overall, scoring 282.5 points out of the 500 games. The program using the parameters learned from the test suite does not do as well. This is not too surprising. Most test suites focus on the tactical abilities of programs. Although tactics are important, the test suites sometimes overemphasize their importance compared to actual game play. As expected, the program using no extensions (C_{100}) performs by far the worst. On the other hand, it is interesting to see how close the other programs performance is, even though they are using quite different weight vectors. The C_{hand}, C_{010}, C_{050} all end up with a similar score. It is a little surprising to see how well the C_{050} version does, intuitively we would have thought it would rank lower. The fact that this version outranks the C_{010} and the C_{ECM} versions shows that in actual game play it is not necessarily good to extend too aggressively, resulting in many irrelevant lines being searched too deeply. Although this might improve the tactical ability of the program, it hurts the positional play and the overall performance.

Unfortunately, we have no way of telling what the optimal weight vector is, and thus we cannot really say how close to optimal the learned weights are. However, based on the above results, we can state with over 90% confidence that the program using the weights learned from game playing performs better than the program using the hand-set weights.[7] The extension-scheme

[7] *Student's t-test* was used to compare the mean of the score-distribution of the two

Table 6.2: Match results.

vs	C_{online}	C_{050}	C_{hand}	C_{010}	C_{ECM}	C_{100}	Points
C_{online}	-	54.5-45.5	51-49	54-46	59.5-40.5	63.5-36.5	282.5
C_{050}	45.5-54.5	-	53.5-46.5	49.5-50.5	53-47	66-34	267.5
C_{hand}	49-51	46.5-53.5	-	50-50	50.5-49.5	64.5-35.5	260.5
C_{010}	46-54	50.5-49.5	50-50	-	48.5-51.5	64.5-35.5	259.5
C_{ECM}	40.5-59.5	47-53	49.5-50.5	51.5-48.5	-	58-42	246.5
C_{100}	36.5-63.5	34-66	35.5-64.5	35.5-64.5	42-58	-	183.5

employed by our test program is a relatively simple one, using only a few parameters. These parameters have been hand-tuned to reasonable values, and thus the opportunity for drastic improvement is small. On the other hand, the benefits of automatic tuning to become increasingly relevant for more sophisticated extension schemes that require the tuning of many parameters.

6.6 Conclusions

In this chapter we introduced a method for automatically tuning search-control parameters in adversary search. By using a cost model to model the search, the learning task can be formulated as a function approximation task, allowing us to use well-established machine learning techniques for determining the most appropriate parameter vector. The learning method was implemented and tested in a strong chess-program, where it learned a parameter vector that outperformed other parameter vectors, including one chosen by a leading computer-chess expert.

Automated tuning of search-control parameters opens up many new opportunities for improved search-control schemes. Traditionally, the effort it takes to hand-tune complex extension schemes — possibly using many disparate parameters — imposes a limitation on how complex the schemes can be in practice. However, by automating the tuning process it is possible to

programs.

experiment with more sophisticated schemes. This is a logical next step. One can envision schemes that use many more move classes, where each class is not only dependent on the phase of the game but also on other positional features. For example, different extensions would apply for open vs. closed positions, or positions where the king is exposed!

Chapter 7

Learning Search Control Offline

Chess is 99% tactics.
- R. Teichmann

In the previous chapter we introduced a general framework for learning search control in adversary search. Based on the framework, we designed and implemented a method for learning search extensions either during online play or by analyzing game positions offline. We experimented with the method in a chess program, where it demonstrated its usefulness by learning search-extension parameters that outperformed all other parameters we tried. On the down-side, it was quite an involved task to incorporate this method into a game-playing program. The subject of this chapter is how to alleviate that problem.

7.1 Introduction

The main drawback of the learning method we introduced in the previous chapter is how intrusive it is. Substantial modifications and additional code are introduced into the game-playing program. To be able to estimate the partial derivatives that drive the learning algorithm, the game-playing program needs to record separate search-depth and node-count information for each search-extension parameter. In practice, these modifications are not always trivial to

implement *efficiently*. It is important not to introduce too much additional computational overhead — otherwise the playing strength of the game-playing program is compromised. This can be a challenging task, especially in a highly optimized program. For one thing, the changes must be implemented deep inside the core of the search engine. To make matters even worse, competitive world-class game-playing programs typically consist of highly optimized code (where software engineering coding principles are mercilessly sacrificed for the sake of additional search speed). This makes code modification even more difficult, and can potentially discourage people from using our method.

In this respect, a less intrusive learning approach is desirable, where the learning module is better separated from the game-playing program itself. Preferably, only minimal (if any) changes should be required to the game-playing program. Fortunately, there is a way of doing this by removing the previous preference that the game-playing program learn during online play. That is, if we limit its use to offline analyzes of game positions, we can separate the learning module from the game-playing engine.

7.2 Offline vs. Online Learning

There are at least two fundamental differences between online and offline learning that affect the way we approach the problem. When learning offline:

- we need not be as concerned with performance overhead incurred by the learning method, and
- we may assume that the correct move is known for each position in our training set (see later).

The above properties simplify the offline learning task considerably.

7.2.1 How to Estimate the Gradient

When designing our online learner, one of the main challenges was how to efficiently estimate the gradient that drives the learning algorithm. We came up with a way of doing this in real-time during the actual search. However, during offline learning the computational overhead is much less of an issue. Therefore, a different and much simpler approach is possible.

Instead of trying to estimate what effects different parameter settings have during a search, it is now possible to obtain this information directly simply by performing a separate search after each parameter change. The offline learning module will call the search engine repeatedly for each game position in the training set, each time using a slightly different parameter setting. This alternative way of estimating the gradient (partial derivatives), although clearly more computationally expensive, has a big benefit of being more modular. This key factor allows us to implement the learning algorithm without modifying the search engine.

7.2.2 How Deep to Search?

In the online version, learning is triggered when critical positions are encountered. However, we do not truly know how deep a search is needed to discover the correct continuation in the critical position (for that matter, we do not even know what the correct continuation is!). This poses an additional challenge in the online learning approach, that is solved by assuming that the program must fully explore a so called critical solution path to avoid playing an inferior move (see previous chapter). This assumption turns out to be reasonable in practice although, as we pointed out, it is not necessarily always true.

On the other hand, during offline learning there is no need to make such an assumption. We can require that all the training samples in the test suite be labeled with information about what the best move is. The game positions

can be analyzed beforehand to determine what the best move is by computer or human analyzes (or a combination thereof). Now, instead of indirectly approximating how deep a search is needed to find the correct move, the offline method simply instructs the game-playing program to explore the position until either the suggested move is found, or an imposed search limit is reached.

7.2.3 Pros and Cons

We are clearly limiting the usefulness of the learning system by restricting its use to only offline play. For example, when meeting a previously unseen opponent in a match spanning a series of games, a program enhanced with online learning capabilities can adjust its play based on experience gained in earlier match games. For example, one respected source for ranking chess programs by their playing strength is the *Swedish Rating List*, published by the Swedish Chess Computer Association (SSDF) both online and in the International Computer Games Association journal. To determine the programs relative playing strength, they are matched against each other, each encounter consisting of a long series of games. Online learning is clearly beneficial in such settings.

On the other hand, if given an opportunity to practice against an opponent beforehand, the learning can equally well be done offline. Lost games can be analysed by a human or a computer player, and a test suite constructed consisting of those positions where a deeper search would have avoided playing the inferior move. These positions could serve as a training data for the offline learner, and the learned search-extension parameters be employed in future encounters with that opponent. Thus, depending on the situation, restricting the learning to offline analysis might not be a serious limitation. Furthermore, the main attraction of the offline learning method is that it can be implemented with only minimal modifications to the game-playing program.

7.3 Architecture

Figure 7.1 shows the basic architecture of the offline learning system. One of the main design objectives behind this architecture is to keep the learning module as separate as possible from the game-playing program, thus minimizing the amount of changes needed to the game-playing program itself. The learning system consists of three main parts: the learning module, the game-playing engine (that is now a separate process) and, finally, a pre-generated database of training examples (where each example consists of a game position and information about what is the best move in that position).

The learning module, which is also the main driver, reads in the game positions from the database (or test suite) and then repeatedly calls the game-playing engine, asking it to solve each of the positions using different search control parameters. In the following subsections we describe each of the components in a more detail.

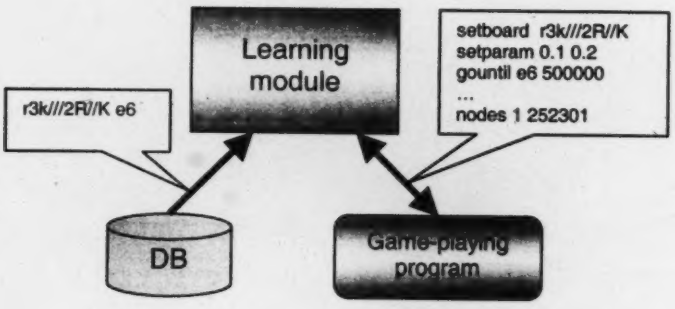


Figure 7.1: The architecture of the offline learning system.

7.3.1 Game-Playing Program

The learning module tells the game-playing program which game position to search, what the best move is in that position, and what search-control parameters to use. In return, the game-playing program informs the learning module about how many nodes were expanded during the search. The only change required to the game-playing program is to augment its command interface to support this interaction. This means implementing the following three commands:

- **setboard** *position*

  Set the current game state to be *position*. The learning module is indifferent to the representation of a game state or position (it simply relays this information from the database), but the game-playing program must understand the format. This command also resets the state of the game engine such that a new search can be performed independently of previous searches (e.g. the transposition table and other history information must be cleared). No return value is expected.

- **setparam** w_1 w_2 ... w_n

  Specify the values of the search-control parameters. The arguments $w_1, ..., w_n$ are real numbers and represent the values that the search-control parameters take. The game-playing program can scale these parameters or map them to integers (if the program's internal representation requires so). No return value is expected.

- **gountil** *move* n

  This command instructs the game-playing program to search the current game position until the program agrees that *move* is the best continuation in the given position, or an imposed search limit of n nodes is

reached. The number of nodes actually searched must be returned, and also a flag indicating whether the suggested move was found by the search. The return string has the following format:

nodes *flag count*

where *flag* is set to 1 if the problem was solved, otherwise 0. The *count* tells how many nodes were expanded by the search (for an unsolved position *count* is the node-count limit n).

We assume that the game-playing program reads its commands from standard input and writes to standard output. Each instruction is followed by a *newline* or *return* character, and the return string is printed to standard output.

Compared to the changes required by the online learner, it is quite trivial to implement the above commands. First of all, they are done to the command interface as opposed to deep inside the search-engine core. Furthermore, many game-playing programs already have commands built-in with similar capabilities, e.g. a command to set up a game position, a command for specifying the value of a (search) parameter, and a command to perform a search. Thus, implementing the above three commands is typically as simple as mapping them onto already supported interface commands.

7.3.2 Learning Module

The learning module is also the main driver. It first spawns off the game-playing program as a separate process. All communication between the learning module and the game-playing engine is done via Unix-domain sockets (pipes) that have been redirected to the game program's standard input/output. Thus, no special process communication support needs to be added to the game-playing engine. The database contains a collection of game positions, where each position is labeled with information about the correct move choice.

Algorithm 13 *LSC*-offline

```
1: for i = 1, N do
2:   w_i ← 1.0
3: end for
4: // Iterate until a sufficiently good w⃗ is found.
5: while not terminate do
6:   nodes ← 0
7:   for i = 1, N do
8:     nodes_i ← 0
9:   end for
10:  for all (p_t, m_t) ∈ T do
11:    nodes = nodes + GetNodecount(p_t, m_t, w⃗, n)
12:    for i = 1, N do
13:      w_i ← w_i + δ
14:      nodes_i ← nodes_i + GetNodecount(p_t, m_t, w⃗, n)
15:      w_i ← w_i − δ
16:    end for
17:  end for
18:  for i = 1, N do
19:    w_i ← w_i − μ Δw_max ((nodes_i − nodes)/nodes)
20:  end for
21:  μ ← Decrease(μ)
22: end while
```

These positions are read from the database and used as training data. Finally, the learning component (shown as Algorithm 13 below) is invoked. The learning, as before, is based on the gradient-descent algorithm. It is almost identical to the learning algorithm we described for the online learning system — the most noticeable difference being how the gradient is found.

The outermost loop iterates until the parameter values converge. The *nodes* variable records the cumulative node count over all the test positions. The variables $nodes_i$, $i = 1, 2, ..., N$ (where N is the number of search-extension parameters) similarly record the cumulative node count for each altered parameter vector. These variables are reset to zero at the beginning of each iteration. Next the algorithm loops over all the training samples, where each sample consists of a game position (p_t) and the best move for that position (m_t). For each position, the game-playing program is called several times:

once using the current parameter vector $\vec{w}$, and then N times using a slightly altered weight vector. At the i-th call, the i-th parameter of the weight vector is modified by a small amount δ. Whereas during the online learning experiment we had to be careful to increment the parameters (to be able to estimate the effects of a change), here it does not matter all that much whether a parameter is increased or decreased. One approach is to modify the parameter in the direction it is currently moving. That is, if the parameter was decreased at the end of the previous learning iteration, then also decrease it slightly when estimating the gradient (and vice versa). The implementation details of how to interact with the game-playing program are hidden in the $GetNodecount(p_t, m_t, \vec{w}, n)$ function. This function sends the three commands described earlier (*setboard* p_t, *setparam* $\vec{w}$, and *gountil* m_t n), and then waits until it receives the expected return string ("nodes ..."). Upon receiving the return string the requested node count information is extracted and returned.

At the end of the learning iteration the parameter vector $\vec{w}$ is modified in the direction opposite to the gradient (the gradient is determined as before by the cumulative node count information). The resulting parameter vector is then used on the subsequent iteration. This continues until a sufficiently good parameter vector is determined. The μ and the Δw_{max} variables are the same as in Chapter 6.

7.4 Experimental Results

We tested the new offline leaning method using an experimental setup similar to the one used in the previous chapter. That is, the weights of the move categories were initialized to 1.0 and allowed to vary within the range [0.1,2.0], an upper limit of half a million nodes was imposed on each search, and the values of μ and Δw_{max} were both set to 1.0. Also the ECM test suite [48] was used as a training input.

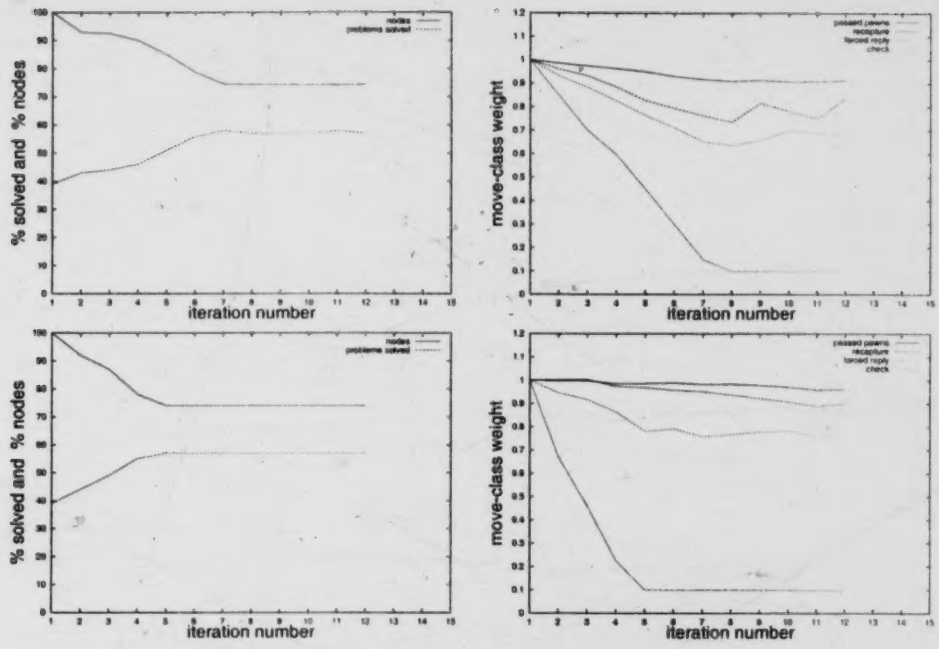


Figure 7.2: Comparison of online (upper) vs. offline (lower) learner.

The result of the experiment is shown in Figure 7.2. The two lower graphs show the new offline learning algorithm, whereas the two upper graphs show the corresponding results from the online learning in the previous chapter (we regenerate the graphs from Figure 6.4 here to make performance comparison between the two methods easier). The left graphs show the search efficiency in terms of number of nodes searched and the number of problems solved, represented by a solid and a dotted line, respectively. The right graphs, on the other hand, demonstrate how the move-class weights evolve. As in Chapter 6 the learning algorithm tries to minimize the number of nodes required to solve the problems, and the increase in the number of problems solved follows indirectly as a side effect. The new offline learner was stopped after 12 iterations because it was clear that the values had already converged.

Beforehand we had hypothesized that given the same training data the offline learning method might outperform the online learner. The reason is

that the offline learner works with accurate node-count information, whereas our online learner uses estimates. What is remarkable about this result is that both learning methods perform equally well! In the end, both solve the same fraction of the problems (57%) and search approximately the same number of nodes (229M). This is reassuring and adds credibility to the claim that the approximate information used by the online learner is sufficiently good for use in practice.

Finally, by comparing the weights learned by the two methods, we see that they differ slightly. However, the difference doesn't affect the performance: both the weights vectors perform equally well. It is quite possible, that both are local optima. More importantly, the check extensions and the forced reply extensions — the two move types that seem to be the most critical to extend on (have the lowest weights) — do converge to close to identical values. The two remaining move-classes, passed-pawns and re-captures only influence the total node count weakly.

7.5 Conclusions

We introduced an alternative method for learning search extensions. One of the main appeals of this approach is that it offers an easy way to tune search-control parameters in almost any search-based game-playing program. Moreover, the game-playing program itself needs only minimal modifications. The changes are as simple as augmenting the game-playing program's interface with the three high-level commands introduced earlier, and the program can then be "plugged" into the learning module.

On the other hand, this ease of use comes with a price. Probably the most serious drawback of the approach introduced here is that it can learn only from analyzing game positions offline, whereas the method we introduced in the previous chapter can also learn during online play. There are two addi-

tional disadvantages with the new offline approach. First, it requires annotated training data, that is, the correct move in each test position must be known. Secondly, it is computationally more expensive than the method we introduced earlier. However, if sufficient computer resources are available, the method is trivial to parallelize. All searches within one iteration can be executed in parallel — the only synchronizing point is at the end of the iteration. Another possible performance optimization is to use shallow searches to estimate the gradient. We will as before do an additional search for each parameter, however, the search can be more shallow because we only need it for determining the growth rate. We will then, as we did in the online method, use our cost model to predict total number of nodes a full search would require.

Which of the two methods for learning search-control parameters is more appropriate depends on the situation. If online learning is required the method introduced in the previous chapter should be used, whereas if offline learning is sufficient, the method introduced in this chapter is to be preferred. Finally, it is worth mentioning that the offline learning method is not assuming that the search-control parameters it is learning are necessarily used for controlling search extensions. On the contrary, the method can be used to learn any type of parameters that influence the search process, for example move-ordering parameters.

Chapter 8

Concluding Remarks

Don't worry, kids. You'll find work. After all, my machine will need strong chess-player programmers. You will be the first.

– Mikhail Botvinnik, ca. 1963 (said to some of his chess students when claiming that his chess program would eventually become the World Champion).

In this chapter we conclude the research by briefly summarizing the main research issues and achievements. Finally, we provide pointers for future research directions.

8.1 Conclusions

In this thesis we investigated selective depth-first expansion of game-trees. Additional full-width search exhibits diminishing returns in terms of an increase in playing strength, so simply searching deeper and deeper in a (semi-) uniform way is not necessarily the best way to harvest the ever increasing hardware speed. Instead, it shows more promise to use the additional computing power to selectively expand the game trees [27]. The question we were primarily concerned with in this work was: *how should game-playing programs best spend their search effort to maximize their move decision quality?*

The first step was to gain additional insight into speculative pruning. Chapter 3 summarizes those findings. Essentially, speculative pruning methods should be concerned with the question: *What is the likelihood of making an*

erroneous pruning decision and, if an erroneous decision is made, how likely is it to affect the principal variation? Surprisingly most pruning methods totally ignore the second part of the question. To answer this question the methods must consider each node in the context of its location in the game tree, instead of looking at each node and the subtree below it in isolation.

This is exactly what the new pruning method we introduce in Chapter 5 does. In both chess and Lines of Action, game-playing programs employing the new *multi-cut* pruning method demonstrate significant increase in playing strength. Furthermore, because the method doesn't rely on game-specific knowledge, it has the potential of being useful in many other games as well.

In Chapter 4 we showed that speculative search, if done in a controlled way, can be used to improve on the search efficiency of $\alpha\beta$-like algorithms, while not affecting the move decision. Over the years several variants of the standard $\alpha\beta$ algorithm have been introduced, each demonstrating a slight improvement in search efficiency over the previous ones. We like to think of our new enhancement, ***uncertainty cutoffs***, as an addition to this line of variants. The improvement in search efficiency, although relatively small, is comparable to the improvement each of the previously published variants demonstrates over the previously established state of the art.

In most game-playing programs, search extensions are necessary for achieving top performance. Unfortunately, parameterizing and tuning the various search-extensions schemes to get a well balanced search is a difficult, tedious, and time consuming task. Extending too aggressively will degrade the overall performance because too much time is spent on exploring irrelevant continuations too deeply at the cost of not reaching sufficient nominal depth. On the other hand, extending too conservatively can result in overlooking important tactics. In Chapter 6 and 7 we introduced a method for learning search-control parameters in adversary search, focusing on search extensions in particular. By

using a cost function to model the search, the learning task can be formulated as a function approximation task, allowing us to use well-established machine learning techniques for determining the most appropriate parameter values. The learning method were tested in the domain of chess, where it learned a parameter vector that outperformed other parameter vectors, including one chosen by a leading computer-chess expert. By automating the tedious tuning process, it becomes feasible to experiment with far more sophisticated extension schemes, using many more parameters.

8.2 Future Work

There are still many unexplored avenues for further research, for example:

- The multi-cut method as described and implemented here is not the only way of reasoning about how likely an erroneous pruning decision is to influence the move decision at the root, and by no means necessarily the best. We hope that this work will pave the road for new speculative pruning methods utilizing such information.

- The multi-cut method does not utilize any game-specific knowledge. We deliberately made this decision to make the approach as domain independent as possible. This is both a strength and a weakness. It is quite possible that the performance of the method can be further enhanced by looking at domain-specific properties.

- The multi-cut method and the singular-extension method complement each other in various ways, while sharing much of the same computational overhead. It is worth looking into combining the two schemes, possible getting the benefits of both while "sharing the cost".

- The method we introduced for learning search-control parameters is not

necessarily specific to adversary search. The only assumption we make here is that there exists a parameterizable depth function that controls how deeply each branch of the tree is expanded. It is definitely worth experimenting with it in other tree-search domains; single-agent search is one that comes to mind.

- In this work we assume that the basic search-control features are given, and the task of the learning method is to tune parameters for deciding their relative importance. Research into methods for discovering or constructing new search-control features for use in two-person games is a difficult task, but might be a rewarding avenue for future research.

Finally, in the same way as a carpenter needs a good set of tools to do his or her work, anyone planning on doing a research in the area of game-tree search requires adequate tools for visualizing and analyzing the search space.

Bibliography

[1] G. M. Adelson-Velskiy, V. L. Arlazarov, and M. V. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.

[2] T. S. Anantharaman. *A Statistical Study of Selective Min-Max Search in Computer Chess.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1990.

[3] T. S. Anantharaman, M. S. Campbell, and F. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[4] J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

[5] D. F. Beal. Experiments with the null move. In *Advances in Computer Chess 5*, pages 65–89. Elsevier Science Publishers B.V., 1989. D. F. Beal (editor).

[6] D. F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990.

[7] D. F. Beal. *The Nature of MiniMax Search.* PhD thesis, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands, 1999.

[8] D. F. Beal and M. C. Smith. Quantification of search extension benefits. *ICCA Journal*, 18(4):205–218, 1995.

[9] D. F. Beal and M. C. Smith. Learning piece values using temporal differences. *ICCA Journal*, 20(3):147–151, 1997.

[10] H. J. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[11] H. J. Berliner. B* probability-based search algorithm. *Artificial Intelligence*, 86(1):97–156, 1996.

[12] J. Birmingham and P. Kent. Tree-searching and tree-pruning techniques. In *Advances in Computer Chess 1*, pages 89–107, Edinburgh, 1977. Edinburgh University Press. M. R. B. Clarke (editor).

[13] Y. Björnsson. YL wins Lines of action tournament. *ICCA Journal*, 23(3):178–179, 2000.

[14] Y. Björnsson and T. A. Marsland. Multi-cut pruning in alpha-beta search. In *Proceedings of The First International Conference on Computers and Games (CG '98). Lecture Notes in Computer Science, special issue Computers and Games*, pages 15–24. Springer Verlag, 1999.

[15] Y. Björnsson and T. A. Marsland. Risk management in game-tree pruning. *Information Sciences Journal*, 122:23–41, 2000.

[16] Y. Björnsson and T. A. Marsland. Learning search control in adversary games. In *Advances in Computer Games 9, H.J. van den Herik and B. Monien (editors)*, pages 157–174. University of Maastricht/University of Paderborn, 2001.

[17] Y. Björnsson and T. A. Marsland. Multi-cut alpha-beta pruning in game-tree search. *Theoretical Computer Science*, 252:177–196, 2001.

[18] Y. Björnsson and T. A. Marsland. Learning control of search extensions. In *Proceedings of the Joint Conference on Information Science (JCIS '02)*, pages 446–449, 2002.

[19] Y. Björnsson, T. A. Marsland, J. Schaeffer, and A. Junghanns. Searching with uncertainty cut-offs. *ICCA Journal*, 29(1):29–37, March 1997. Appeared also in book Advances in Computer Chess 8, H.J. van den Herik and Jos Uiterwijk (editors).

[20] Y. Björnsson and M. H. M. Winands. YL wins Lines of action tournament. *ICGA Journal*, 24(3):180–181, 2001.

[21] I. Bratko and M. Gams. Error analysis of the minimax principle. In *Advances in Computer Chess 3*, pages 1–16. Pergamon Press, 1981.

[22] D. M. Breuker. *Memory versus Search in Games.* PhD thesis, Universiteit Maastricht, Maastricth, The Netherlands, 1998.

[23] A. L. Brudno. Bounds and valuations for abridging thee search of estimates. *Problems of Cybernetics*, 10:225–241, 1963. Originally appeared in *Problemy Kibernetiki*, vol. 10, pp. 141-150 (in Russian).

[24] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[25] M. Buro. Experiments with multi-probcut and a new high-quality evaluation function for Othello. In *Games in AI Research*, pages 77–96, Maastricht, The Netherlands, 2000. Universiteit Maastricht. H.J. van den Herik, H. Iida (editors).

[26] M. Buro. Towards opening book learning. In *Games in AI Research*, pages 47–54, Maastricht, The Netherlands, 2000. Universiteit Maastricht. H.J. van den Herik, H. Iida (editors).

[27] M. S. Campbell, A. J. Hoane Jr., and F. Hsu. DEEP BLUE. *Artificial Intelligence*, 134:57–83, 2002.

[28] T. Cazenave. Generation of patterns with external conditions for the game of Go. In *Advances in Computer Games 9, H.J. van den Herik and B. Monien (editors)*, pages 275–293. University of Maastricht/University of Paderborn, 2001.

[29] D. Dailey and C. E. Leiserson. Using CILK to write multiprocessor chess programs. In *Advances in Computer Games 9, H.J. van den Herik and B. Monien (editors)*, pages 25–52. University of Maastricht/University of Paderborn, 2001.

[30] C. Donninger. Null move and deep search: Selective serach heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, 1993.

[31] T. A. Estlin and R. J. Mooney. Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI-97)*, 1997.

[32] R. Feldmann. Fail-high reductions. In *Advances in Computer Chess 8*, June 1997.

[33] J. P. Fishburn. *Analysis of Speedup in Distributed Algorithms*, volume 14 of *Computer Science: Distributed Database Systems*, chapter 4 – Parallel Alpha-Beta Search, pages 11–54. UMI Research Press, 1984. Revision of thesis (PhD) – University of Wisconsin, Madison, 1981.

[34] J. Fürnkranz. Machine learning in computer chess: The next generation. *ICCA Journal*, 19(3):147–161, 1996.

[35] J. Fürnkranz. *Machines That Learn To Play Games*, chapter Machine Learning in Games: A Survey, pages 11–59. Nova Science Publishers, Inc., Huntington, New York, 2001. J. Fürnkranz and M. Kubat (editors).

[36] G. Goetsch and M. S. Campell. Experiments with the null-move heuristic. In T.A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 159–168, New York, 1990. Springer-Verlag.

[37] R. D. Greenblatt, D. E. Eastlake III, and S. D. Crocker. The GREENBLATT chess program. In *Proceedings of the Fall Joint Computer Conference*, pages 801–810. AFIPS Press, 1967.

[38] K. Greer. Computer chess move-ordering schemes using move influence. *Artificial Intelligence*, 120:235–250, 2000.

[39] E. A. Heinz. *Scalable Search in Computer Chess*. Vieweg Verlag, Germany, 2000.

[40] E. A. Heinz. Self-play, deep search and diminishing returns. *ICGA Journal*, 24(2):75–79, June 2001.

[41] R. M. Hyatt. Crafty - chess program. 1996. ftp.cis.uab.edu/pub/hyatt.

[42] R. M. Hyatt. Book learning a methodology to tune an opening book automatically. *ICCA Journal*, 22(1):3–12, 1999.

[43] A. Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21:14–32, March 1997.

[44] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[45] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[46] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. reprinted in Chapter 6 of *Expert Systems, A Software Methodology for Modern Applications*, P.G. Raeth (Ed.), IEEE Computer Society Press, Washington D.C., 1990, pp. 380-389.

[47] V. Koscis and J. Uiterwijk. Learning move ordering in chess. In *Proceedings of the CMG Sixth Computer Olympiad Computer Games Workshop*, July 2001.

[48] N. Krogius, A. Livsic, B. Parma, and M. Taimanov. *Encyclopedia of Chess Middlegames*. 1980.

[49] D. Levy, D. Broughton, and M. Taylor. The sex algorithm in computer chess. *ICCA Journal*, 12(1):10–21, 1989.

[50] T. A. Marsland. Relative efficiency of alpha-beta implementations. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 763–766, Karlsruhe, Germany, August 1983.

[51] T. A. Marsland. *Computer Chess and Search*, pages 224–241. Encyclopedia of Artificial Intelligence. John Wiley & Sons, Inc., Chicester, UK, 1992.

[52] T. A. Marsland. *Computer Chess and Search*, pages 224–241. Encyclopedia of Artificial Intelligence. John Wiley & Sons, Inc., Chicester, UK, 1992.

[53] D. A. McAllester. Conspiracy numbers for min-max searching. *Artificial Intelligence*, 35:287–310, 1988.

[54] D. A. McAllester and D. Yuret. Alpha-beta-conspiracy search, 1993. URL: http://www.research.att.com/~dmac/abc.ps.

[55] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-based Approach*. Kluwer Academic Publishers, Boston, MA, 1988.

[56] T. M. Mitchell. *Machine Learning*, pages 92–94. WCB McGraw-Hill, 1997.

[57] T. M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[58] A. Newell, J. C. Shaw, and H. A. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, pages 320–335, October 1958. Reprinted in [59].

[59] A. Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought*, pages 39–70, 1963.

[60] A. J. Palay. *Searching with probabilities*. PhD thesis, Carnegie-Mellon Univ., Boston, Mass., 1983. See also (1985), book same title, Pitman.

[61] J. Pearl. Asymptotical properties of minimax trees and game searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[62] M. A. Perez. Representing and learning quality-improving search control knowledge. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 382–390, Bari, Italy, July 1996.

[63] Aske Plaat. *Research Re: search & Re-search.* PhD thesis, Tinbergen Institute and Department of Computer Science, Rotterdam, Netherlands, June 1996.

[64] A. Reinefeld. An improvement to the Scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[65] A. Reinefeld and T. A. Marsland. A quantitative analysis of minimal window search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 951–953, 1987.

[66] F. Reinfeld. *1001 Brilliant Ways to Checkmate.* Sterling Publishing Co., New York, N. J., 1955. Reprinted by Melvin Powers Wilshire Book Company.

[67] S. Sackson. *A Gamut of Games.* Pantheon Books, New York, 1982.

[68] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[69] A. L. Samuel. Some studies in machine learning using the game of checkers II. *IBM Journal of Research and Development*, 11(6):601–617, 1965.

[70] J. Schaeffer. *Experiments in Search and Knowledge.* PhD thesis, Department of Computing Science, University of Waterloo, Canada, 1986. Available as University of Alberta technical report TR86-12.

[71] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(1):1203–1212, November 1989.

[72] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43:67–84, 1990.

[73] T. Scherzer, L. Scherzer, and D. Tajden. *Computers, Chess, and Cognition*, chapter 12 — Learning in Bebe, pages 197–216. Springer-Verlag, New York, NY, 1990.

[74] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[75] D. J. Slate and L. R. Atkin. *Chess Skill in Man and Machine*, chapter 4. CHESS 4.5 – Northwestern University Chess Program, pages 82–118. Springer-Verlag, New York, NY, 1977.

[76] M. E. Stickel and W. M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 1073–1075, Los Angeles, California, 1985.

[77] G. C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[78] D. Takahashi. Games get serious. *Red Herring: The Business of Technology*, (87):64–70, December 2000.

[79] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press, Princeton, 1944. Second edition.

[80] D. E. Wilkins. Using knowledge to control tree searching. *Artificial Intelligence*, 18:1–51, 1982.

[81] M. H. M. Winands. Analysis and implementation of Lines of action. Master's thesis, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands, 2000.

[82] C. Ye and T. A. Marsland. Experiments in forward pruning with limited extensions. *ICCA Journal*, 15(2):55–66, 1992.

[83] A. L. Zobrist. A hashing method with applications for game playing. Technical Report 88, University of Wisconsin at Madison, Madison, WI, 1970. Reprinted in the *ICCA Journal* 13, no. 2 (1990), pp. 69-73.

Appendix A

Game-Tree Viewer

The *Game Tree Viewer* is a tool we developed for visualizing game trees. Game programs now-a-days expand huge search trees, typically consisting of millions of nodes. To verify the correctness of the search, programmers try to spot inconsistencies by analyzing logs generated by their programs. However, given the size of these logs this approach is becoming increasingly unfeasible. A graphical tool for viewing the logs makes the task more manageable.

We used the game-tree viewer both to understand better the search process, and to help track down errors. One example of its usefulness is that a serious search bug that had gone undetected in our chess program (THE TURK) for almost two years, was spotted almost immediately when the expanded search trees were viewed graphically. The game-tree viewer toolkit consists of two main parts: a library for writing game trees to a file, and a viewer to interpret and view graphically the data in the file. The library code is linked into the game-playing program, whereas the viewer is a stand-a-lone application.

A.1 Viewer

The viewer is written in *Tcl/Tk* using extensions provided by the *Tix* shell. The core part of the viewer does not include any game-specific code. Instead it relies on plug-ins for interpreting the data, thus allowing a modular design.

For example, we have added plug-ins for several different games, e.g. chess, Chinese-chess, Othello and Sokoban (a single-agent puzzle game).

The viewer displays a single search tree at a time. For example, a program that uses iterative-deepening at the root performs multiple searches (each progressively deeper). It would thus generate several game-tree files (one for each search). Figure A.1 shows an example screen-shot of the game-tree viewer. In this particular example the viewer is used to browse a shallow search tree generated by a chess program. The left half of the screen displays the search tree in an hierarchical fashion. The look-and-feel is almost identical to how directory structures are viewed using file managers. For example, one can expand and collapse the branches as needed. This is important because — as mentioned above — a typical search tree is generally huge. The viewer stores in memory only the branches of the tree that are currently expanded in the viewer, once a branch is collapsed the memory is returned. To get a faster access to the game-tree file during the interactive browsing, an index into the file is created at the time of start-up. This is done in the background such that the user can immediately start browsing the parts of the tree already indexed. Also, to further enhance the access time, a script is provided that does a one time pre-processing of the game-tree file. The script reorganizes the entries in the file such that the nodes are ordered in a breath-first fashion (as opposed to the depth-first fashion generated by the game-playing program). Using these optimizations the viewer can effortlessly handle game-tree files consisting of several millions of nodes.

In the right half of the screen a selected game position is displayed. A position is selected by clicking on the corresponding node in the tree hierarchy (shown as highlighted). A display located beneath the board shows various logistics about the search path leading from the selected game position to the root of the tree. For each position on the path the display shows the

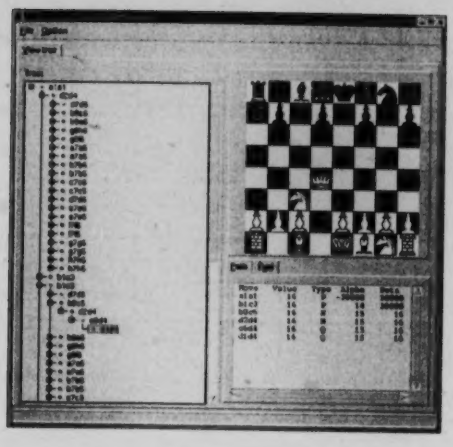


Figure A.1: Screen shot of the game-tree viewer.

move leading to that position, the value of the node, the type of the node (P=principal-variation node, N=minimal-window node, Q=quiescence node), and the lower and upper bounds used when searching the node. For example, from this small example one can see that the move *d2d4* was in the start-position initially considered as the main line (the bullets in front of the moves have a different color indicating the type of the move, e.g. principal variation moves are displayed as red). However, when the move *b1c3* was considered, it failed-high, was re-searched (the move appears twice in the move list), and yielded a new principal variation. We can also see that during the re-search the search window was tightened from 14 to 15, most likely by retrieving better information from the transposition table.

A.2 Library

The library code is written in the *C* programming language and is linked into the game-playing program. It is responsible for recording the tree traversal and for incrementally writing the game tree out to a file. It consists of the following functions:

- *int gtv_New(void)*

  This function is called once to allocate memory and to initialize various internal data structures. No other game-tree library function can be invoked prior to this function being called.

- *int gtv_startTree(int no, char startpos[])*

  Create a new game-tree file identified by the number *no* (the file name will be tree_*no*.gtv). The parameter *startpos* is a string encoded in FEN notation indicating the start position of the search.

- *int gtv_enterNode(Move move, Value alpha, Value beta, int type)*

  Add a new node to the tree. The last move leading to this node is *move*, the search bounds used are *alpha* and *beta*, and the node is of type *type*.

- *void gtv_exitNode(Move move)*

  As the search backtracks this function is used to record the return value.

- *void gtv_stopTree(void)*

  Close the game-tree file.

- *void gtv_Delete(void)*

  Called once at the very end to perform cleanup.

The above function calls are embedded into the search routines of the game-playing program. As an example we show how to embed the functions into a Principal Variation Search algorithm using an iterative-deepening driver at the root (the pseudo code for the *MWS* function is not shown, but the game-tree library calls are embedded in a similar way as in *PVS*). Furthermore, these functions are implemented as macros and are only activated if the code is compiled with a *USE_GTV* flag defined. If not, the pre-processor will omit the *gtv* code.

Algorithm 14 $ID - PVS(P, d, maxdepth)$

```
1: for depth ← 1 to maxdepth do
2:   gtv_startTree(depth, toFENString(P))
3:   v ← PVS(P, depth, −∞, ∞)
4:   gtv_stopTree()
5: end for
6:
7: function PVS(P, d, α, β)
8: gtv_enterNode(lastMove(P), α, β, pvnode)
9: if d ≤ 0 or isTerminal(P) then
10:   v = evaluate(P)
11:   gtv_exitNode(v)
12:   return v
13: end if
14: M ← generateMoves(P)
15: make(P, m_1)
16: best ← −PVS(P, d − 1, −β, −α)
17: retract(P, m_1)
18: if best ≥ β then
19:   gtv_exitNode(best)
20:   return best
21: end if
22: lower ← max(α, best)
23: for all m_i ∈ M|i > 1 do
24:   make(P, m_i)
25:   v ← −MWS(P, d − 1, −lower)
26:   if v > lower and v < β then
27:     v ← −PVS(P, d − 1, −β, −v)
28:   end if
29:   retract(P, m_i)
30:   if v > best then
31:     best ← v
32:     if best ≥ β then
33:       gtv_exitNode(best)
34:       return best
35:     end if
36:     lower ← max(α, best)
37:   end if
38: end for
39: gtv_exitNode(best)
40: return best
```

Appendix B

Estimating $B(\vec{w})$ - Example

In Chapter 6 the introduced a method for learning search control during actual game play. One of the challenges using this method is to estimate efficiently in real-time the effects changing each of the search control parameters has on the growth rate of the search. In the aforementioned chapter we outlined the technique used to do the estimation; in here we illustrate the technique using actual data.

Figure B.1 shows a search tree expanded using a depth threshold of 2.0. In this example there are only two move-classes; the first has fractional-ply weight of 0.4 (pictured using dotted lines) and the second a weight of 1.0 (pictured using solid lines). We use the vector $\vec{w} = \{0.4, 0.1\}$ to represent these weights. The number besides each node shows the depth of the node. As soon as the depth equals (or exceeds) the depth threshold the node is evaluated and the search backtracks. A count of the total number of nodes expanded is also kept. In this example 17 nodes are expanded, thus the growth rate of the search is

$$B(\vec{w})^{2.0} = 17 \implies B(\vec{w}) = 4.123$$

The problem we face is that we also need to simultaneously approximate the growth rate of the search if the parameter vectors $\vec{w}_1 = \{0.4 + \delta, 1.0\}$ and $\vec{w}_2 = \{0.4, 1.0 + \delta\}$ were instead used to expand the tree (needed for calculating the gradient). This is done by recording for each parameter vector a separate

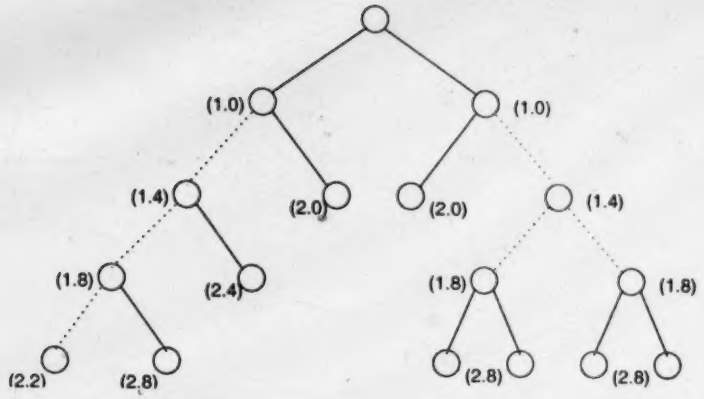


Figure B.1: Depth of nodes in a game tree.

depth and count of number of nodes expanded. Figure B.2 demonstrates this process using $\delta = 0.1$.

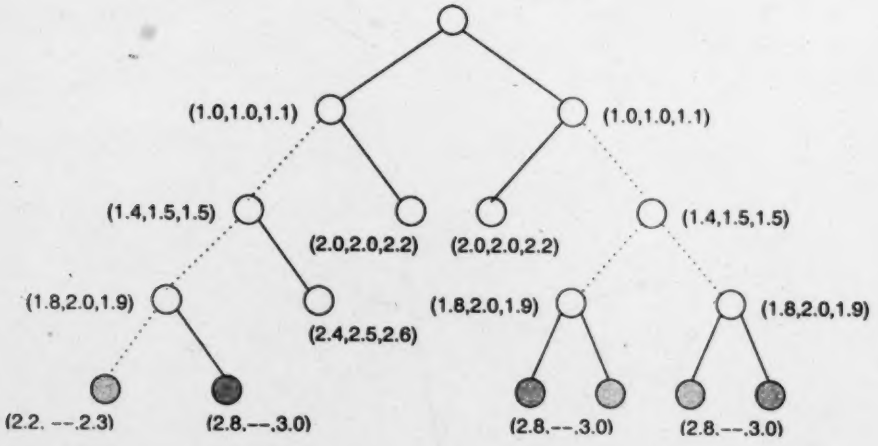


Figure B.2: Multiple depths of nodes in a game tree.

Instead of only one depth, each node has now three depths associated with it. Each of the depths is recorded by a different weight vector, that is, $\vec{w} = \{0.4, 1.0\}$, $\vec{w}_1 = \{0.5, 1.0\}$ and $\vec{w}_2 = \{0.4, 1.1\}$, respectively. In the figure, the leftmost depth of a triplet is the same as used in Figure B.1. As before it determines when to stop expanding the branches in the search tree. The two

other auxiliary depths are used only for deciding whether or not to include a node in the total node count for the alternative weight vectors. That is, as soon as a depth reaches the threshold limit, the search stops counting nodes in the subtree below as being expanded by that weight vector. For example, in the figure the shaded nodes are not included in the total node count for vector $\vec{w}_1$, because the $\vec{w}_1$ depth has already reached 2.0. The rationale is that if we were using that weight vector to expand the tree, this branch would not be explored this deeply. On the other hand, in this example the $\vec{w}_2$ weights expand exactly the same tree as $\vec{w}$. Thus,

$$B(\vec{w}_1)^{2.0} = 11 \implies B(\vec{w}_1) = 3.317$$

$$B(\vec{w})^{2.0} = 17 \implies B(\vec{w}_2) = 4.123$$

This is, of course, only an approximation of the size of the actual trees explored if the alternative weight vectors were instead used. The reason for this is that different evaluations would be backed up the tree, possibly causing different branches of the tree to be explored. However, this technique can be used with little overhead during the search, and works well in practice.

Appendix C

Gradient of Cost Model

The cost model used in Chapter 6 has the basic form:

$$C(p, \vec{w}, d) = B(p, \vec{w})^d$$

Furthermore, we used the partial derivative of

$$C(p, \vec{w}, D(S, \vec{w}))$$

in our learning algorithm. Below we showed how we derived the partial derivatives:

$$\begin{aligned}
\frac{\partial C(p, \vec{w}, D(S, \vec{w}))}{\partial w_i} &= \frac{\partial (B(p, \vec{w})^{D(S, \vec{w})})}{\partial w_i} \\
&= \frac{\partial (e^{\ln (B(p, \vec{w})^{D(S, \vec{w})})})}{\partial w_i} \\
&= \frac{\partial (e^{D(S, \vec{w}) \ln B(p, \vec{w})})}{\partial w_i} \\
&= e^{D(S, \vec{w}) \ln B(p, \vec{w})} \frac{\partial (D(S, \vec{w}) \ln B(p, \vec{w}))}{\partial w_i} \\
&= C(p, \vec{w}, D(S, \vec{w})) \frac{\partial (D(S, \vec{w}) \ln B(p, \vec{w}))}{\partial w_i} \\
&= C(p, \vec{w}, D(S, \vec{w})) \left(D(S, \vec{w}) \frac{\partial (\ln B(p, \vec{w}))}{\partial w_i} + \frac{\partial (D(S, \vec{w}))}{\partial w_i} \ln B(p, \vec{w}) \right) \\
&= C(p, \vec{w}, D(S, \vec{w})) \left(D(S, \vec{w}) \frac{1}{B(p, \vec{w})} \frac{\partial B(p, \vec{w})}{\partial w_i} + \frac{\partial (D(S, \vec{w}))}{\partial w_i} \ln B(p, \vec{w}) \right) \\
&= C(p, \vec{w}, D(S, \vec{w})) \left(\frac{D(S, \vec{w})}{B(p, \vec{w})} \frac{\partial B(p, \vec{w})}{\partial w_i} + \frac{\partial (D(S, \vec{w}))}{\partial w_i} \ln B(p, \vec{w}) \right)
\end{aligned}$$

Appendix D

Test Suites

This appendix lists the various test positions used in our experiments. Some of the test suites we used are far too large to reproduce here, in which case, we provide a reference to where they are published. The first three chess test suites listed consist mainly of tactical game positions. Whereas such test suites are good as a first line of defense for testing the soundness of new ideas, ultimately, playing games is the most reliable way of measuring how different enhancements affect a program's playing strength. The next two suites consist of the opening positions used as a starting point in our computer vs. computer matches. A different starting positions was used for each encounter to prevent the programs from repeatedly playing the same game.

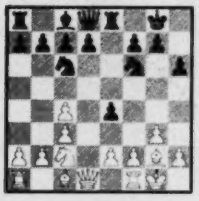
To our knowledge, there do not yet exists any standard opening positions for the game Lines-of-Action. In our experiments we generated from the initial game state all possible game positions 2-ply into the game (one move for each side). This results in 311 different start positions, after symmetrical positions were excluded.

In the following sections we show diagrams of the game position in each of the test suites. The caption above each diagram shows the number of the test position (within the test suite) and which side there is to move in that position.

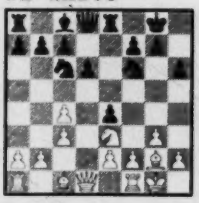
D.1 Plaat Test Positions

These positions were used by Aske Plaat in his thesis work [63]. They are taken from the same game, thus providing positions representative of different stages of a game.

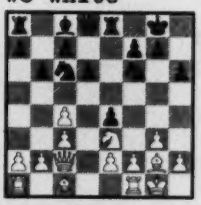
#1 White



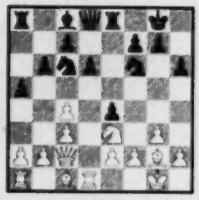
#2 White



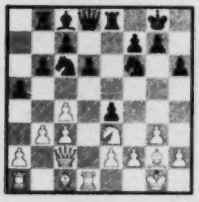
#3 White



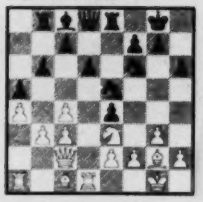
#4 White



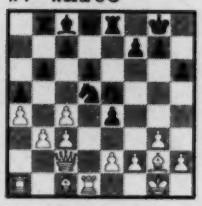
#5 White



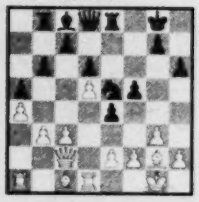
#6 White



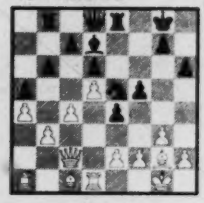
#7 White



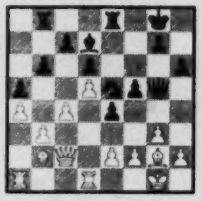
#8 White



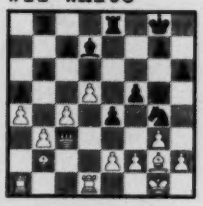
#9 White



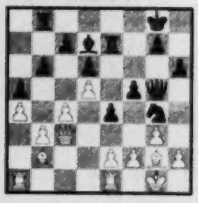
#10 White



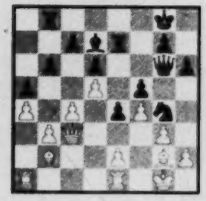
#11 White



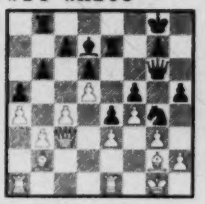
#12 White



#13 White

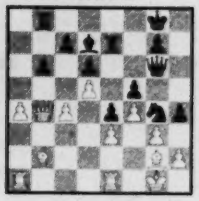


#14 White

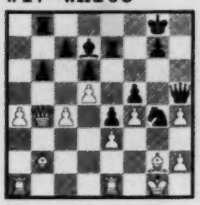


#15 White

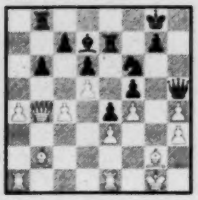
#16 White



#17 White

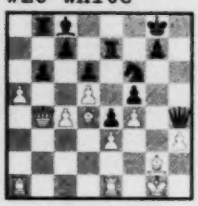


#18 White



#19 White

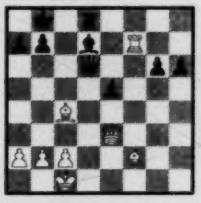
#20 White



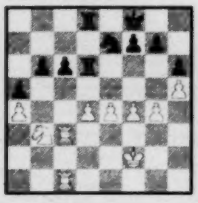
D.2 Bratko-Kopec Test Suite

The Bratko-Kopec test suite [21] consists of a mixture of tactical and positional chess positions, 24 in total.

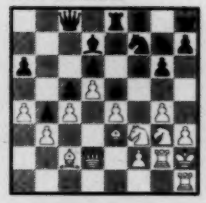
#1 Black



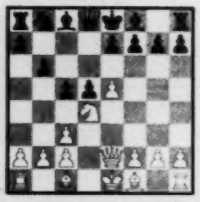
#2 White



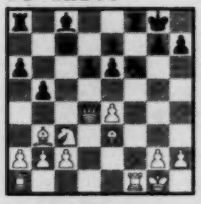
#3 Black



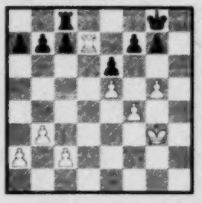
#4 White



#5 White



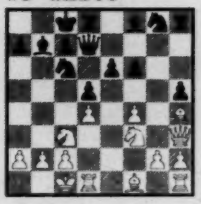
#6 White



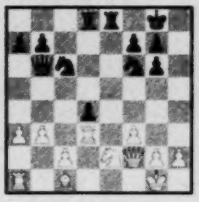
#7 White

#8 White

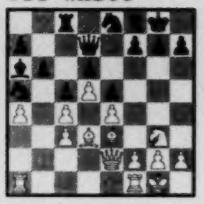
#9 White



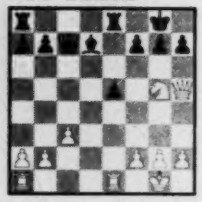
#10 Black



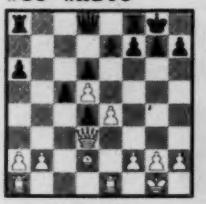
#11 White



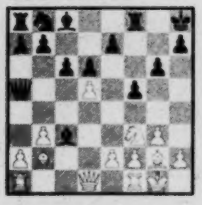
#12 Black



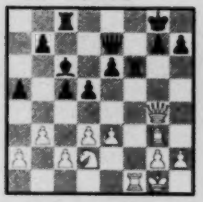
#13 White



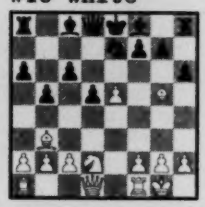
#14 White



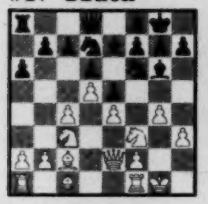
#15 White



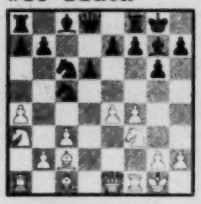
#16 White



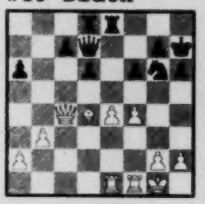
#17 Black



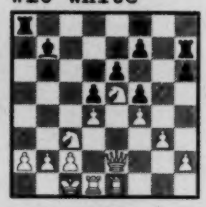
#18 Black



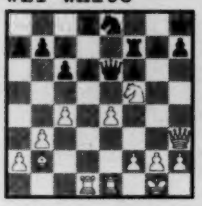
#19 Black



#20 White

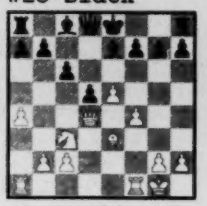


#21 White

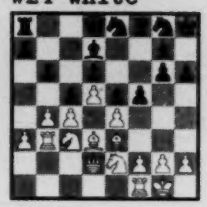


#22 Black

#23 Black



#24 White



D.3 1001BWC Test Suite

This test suite consist of 1001 checkmate problems, taken from Reinfeld's book *1001 Brilliant Ways to Checkmate* [66]. It is far too big to list here but the book has been reprinted many times and is still widely available.

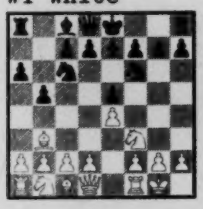
D.4 ECM Test Suite

This test suite is taken from the *Encyclopedia of Chess Middlegames* book by Krogius et al. [48]. It consists or 879, mainly tactical, middle-game positions. Many of the problems in this test suite are quite challenging, for both humans and chess-playing programs.

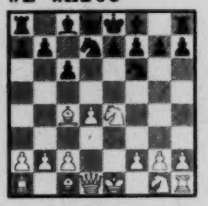
D.5 Opening Position Test Suite

This is a home-cooked test suite we used in our early experiments (before we received Dailey's opening test suite, listed next). It consists of 40 well-known chess opening positions. We used this suite as starting positions in THE TURK self-play matches described in Chapter 5.

#1 White

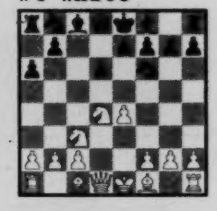


#2 White

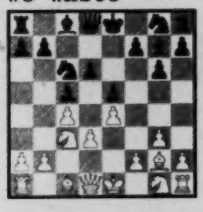


#3 White

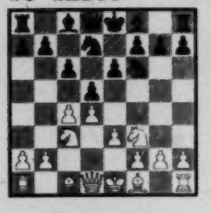
#4 White



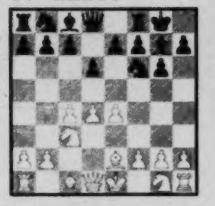
#5 White



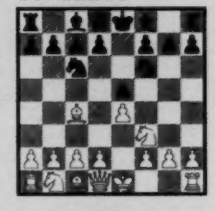
#6 White



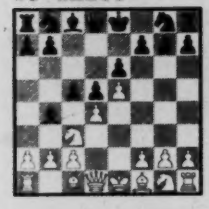
#7 White



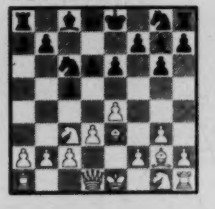
#8 White



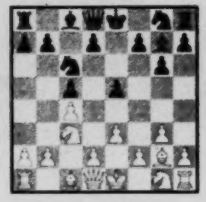
#9 White



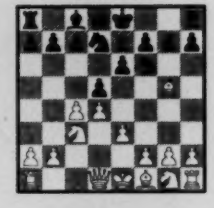
#10 White



#11 White

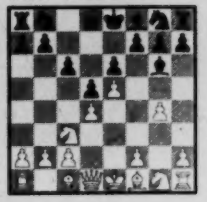


#12 White

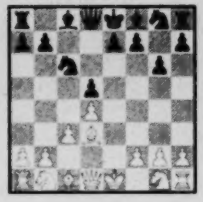


#13 White

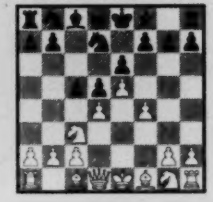
#14 White



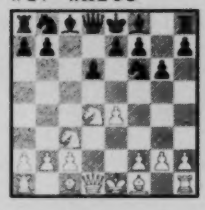
#15 White



#16 White

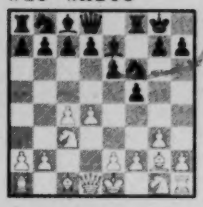


#17 White



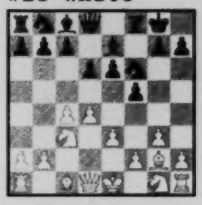
#18 White

#19 White



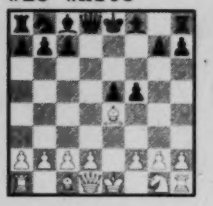
#20 White

#21 White

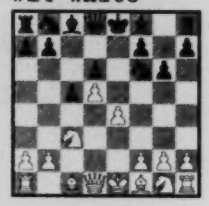


#22 White

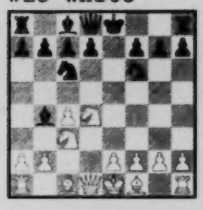
#23 White



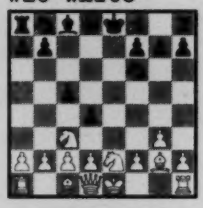
#24 White



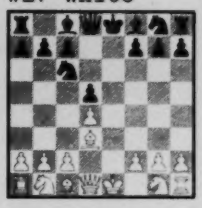
#25 White



#26 White

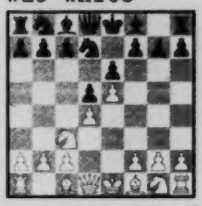


#27 White



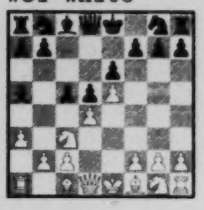
#28 White

#29 White

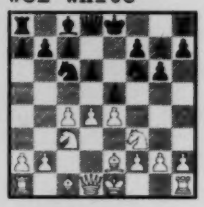


#30 White

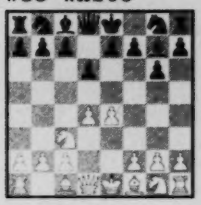
#31 White



#32 White

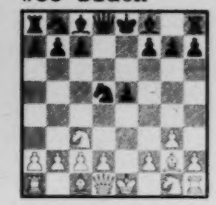


#33 White



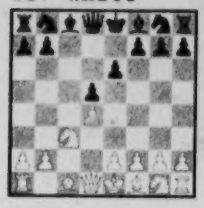
#34 White

#35 Black

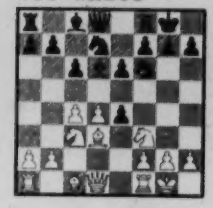


#36 White

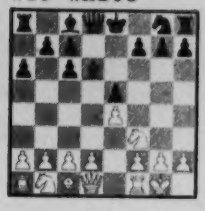
#37 White



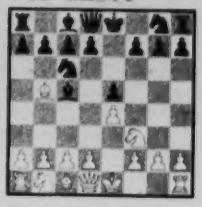
#38 White



#39 White



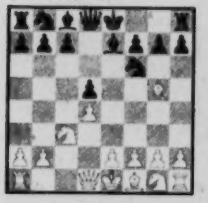
#40 White



D.6 Dailey Opening Test Suite

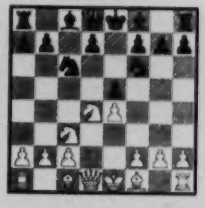
Don Dailey, a co-author of the massively parallel chess program CILK [29], assembled this collection of well-established chess opening positions. He generously made it available to us, for which we are grateful. Not only is it more extensive than the opening suite we used previously, but it is also highly desirable to use independently produced test data when conducting research.

#1 White

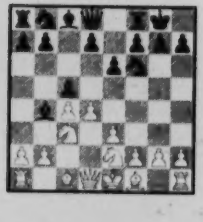


#2 White

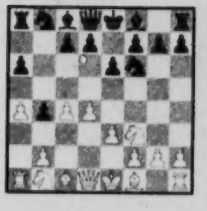
#3 White



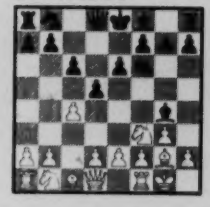
#4 White



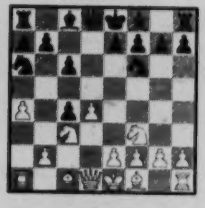
#5 White



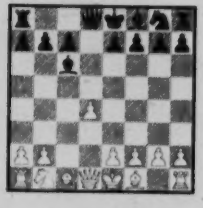
#6 White



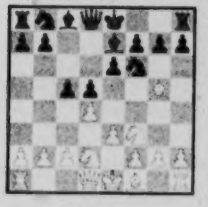
#7 White



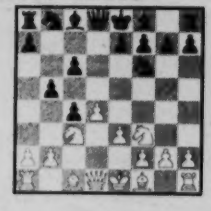
#8 White



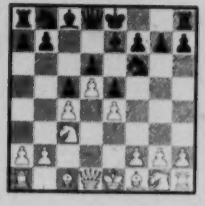
#9 White



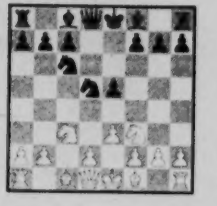
#10 White



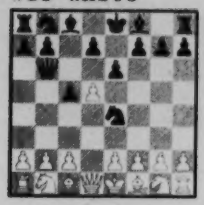
#11 White



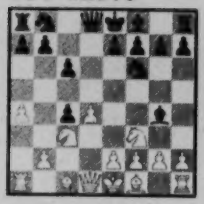
#12 White



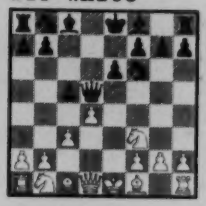
#13 White



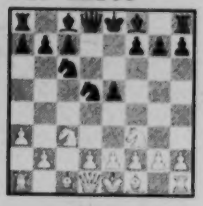
#14 White



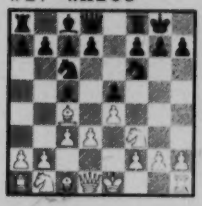
#15 White



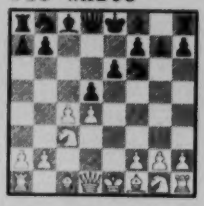
#16 White



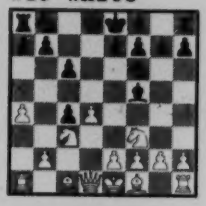
#17 White



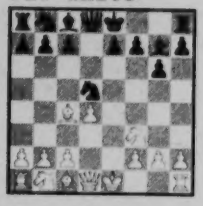
#18 White



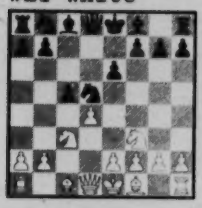
#19 White



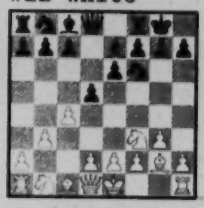
#20 White



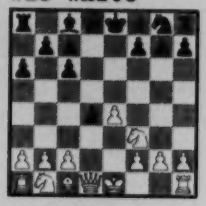
#21 White



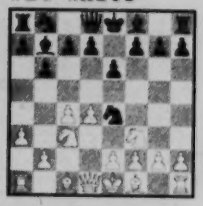
#22 White



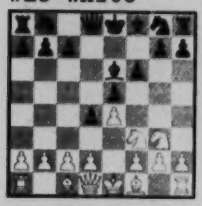
#23 White



#24 White

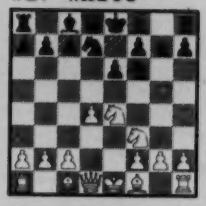


#25 White

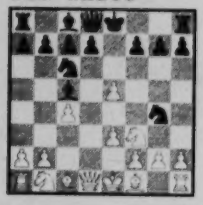


#26 White

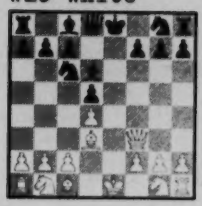
#27 White



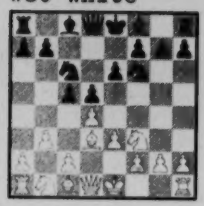
#28 White



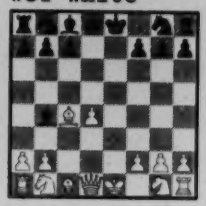
#29 White



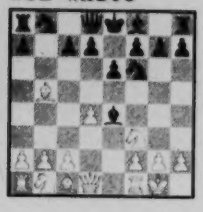
#30 White



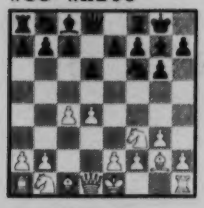
#31 White



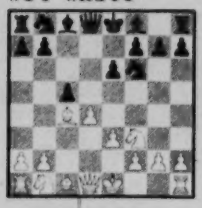
#32 White



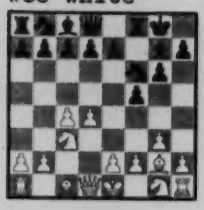
#33 White



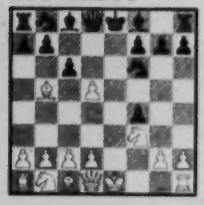
#34 White



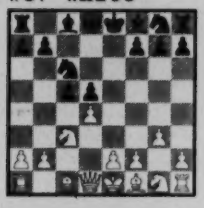
#35 White



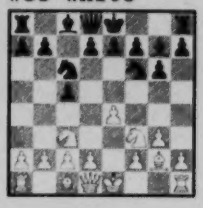
#36 White



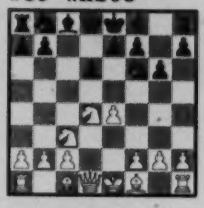
#37 White



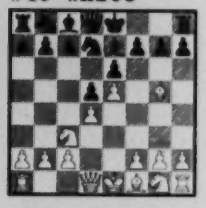
#38 White



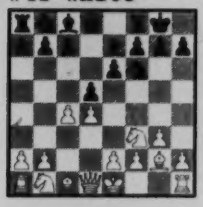
#39 White



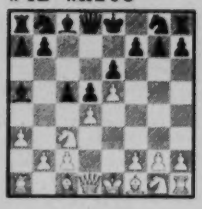
#40 White



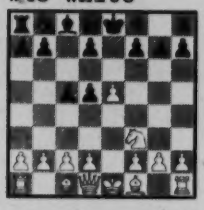
#41 White



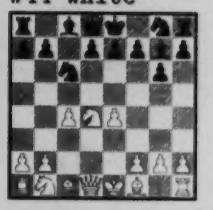
#42 White



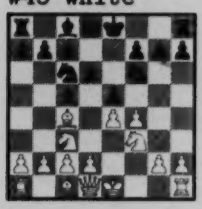
#43 White



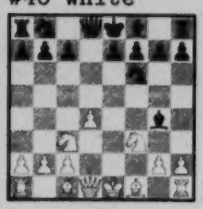
#44 White



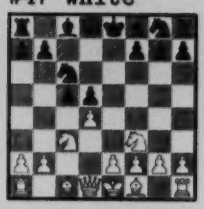
#45 White



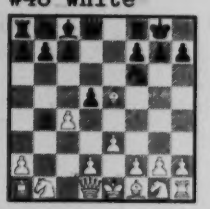
#46 White



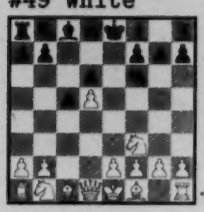
#47 White



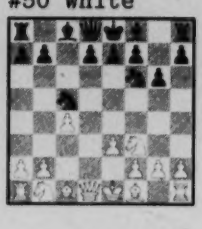
#48 White



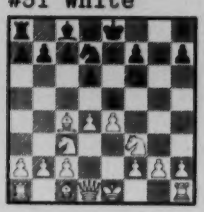
#49 White



#50 White



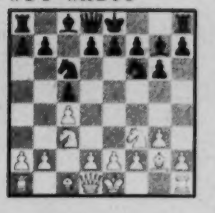
#51 White



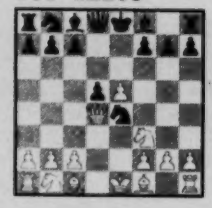
#52 White

#53 White

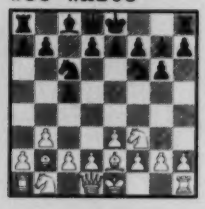
#54 White



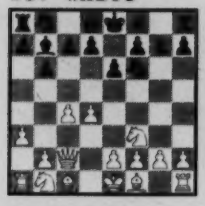
#55 White



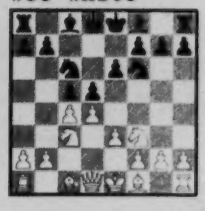
#56 White



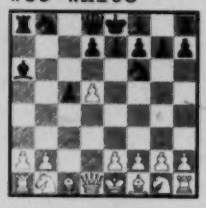
#57 White



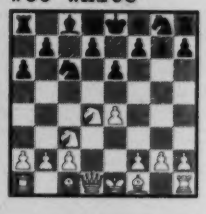
#58 White



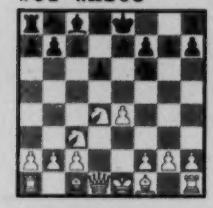
#59 White



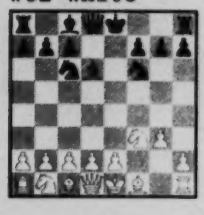
#60 White



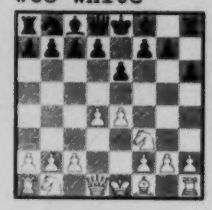
#61 White



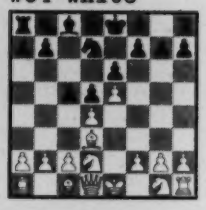
#62 White



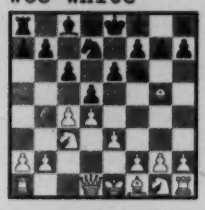
#63 White



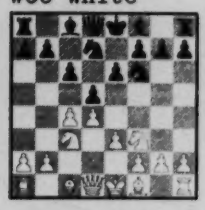
#64 White



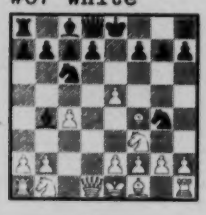
#65 White



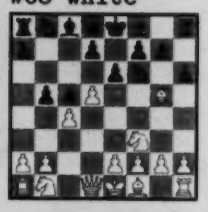
#66 White



#67 White

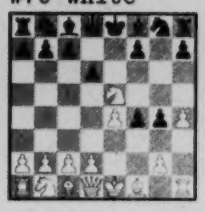


#68 White

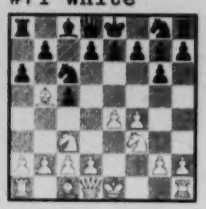


#69 White

#70 White

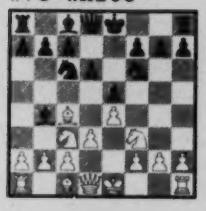


#71 White



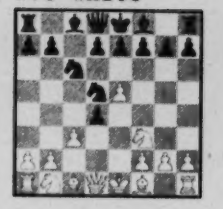
#72 White

#73 White

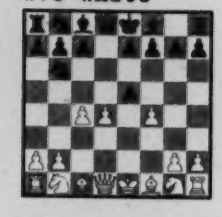


#74 White

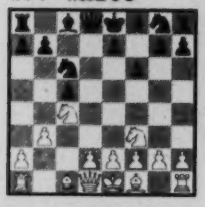
#75 White



#76 White

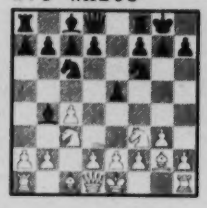


#77 White

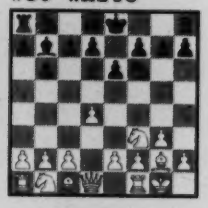


#78 White

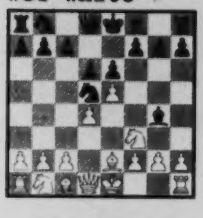
#79 White



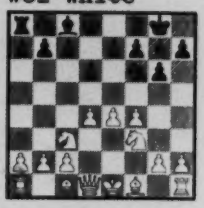
#80 White



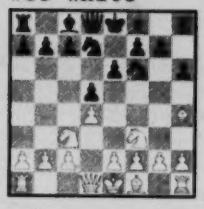
#81 White



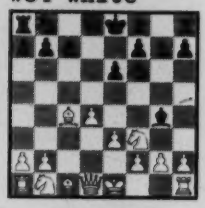
#82 White



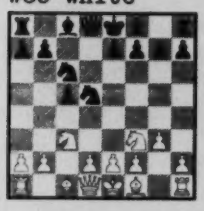
#83 White



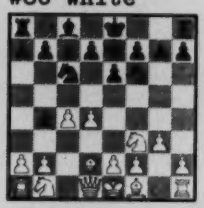
#84 White



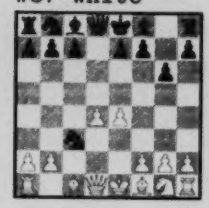
#85 White



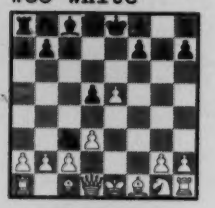
#86 White



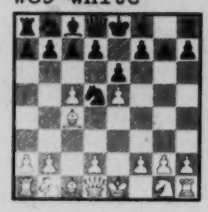
#87 White



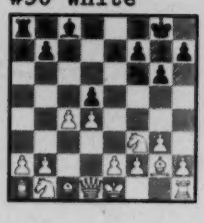
#88 White



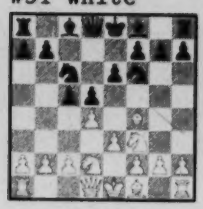
#89 White



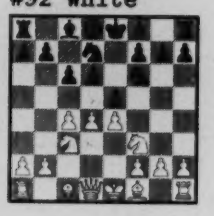
#90 White



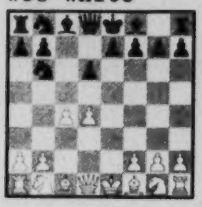
#91 White



#92 White

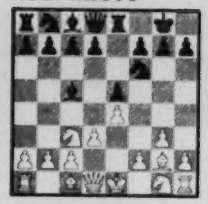


#93 White

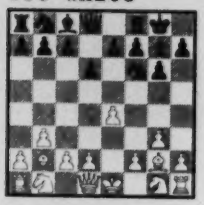


#94 White

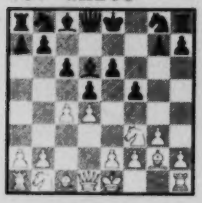
#95 White



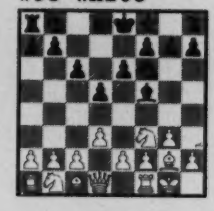
#96 White



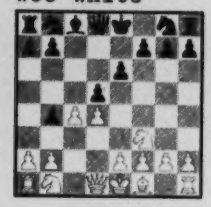
#97 White



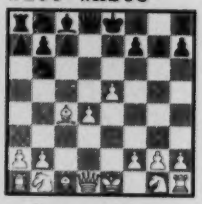
#98 White



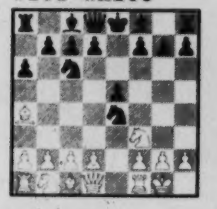
#99 White



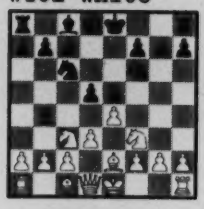
#100 White



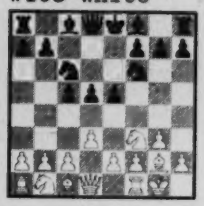
#101 White



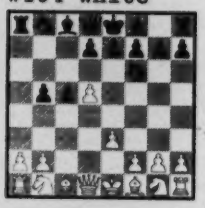
#102 White



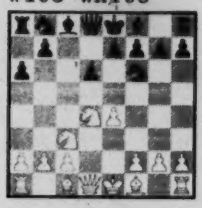
#103 White



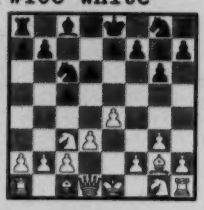
#104 White



#105 White

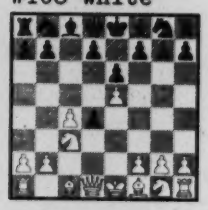


#106 White

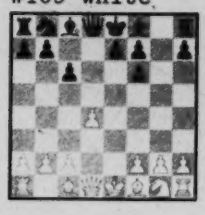


#107 White

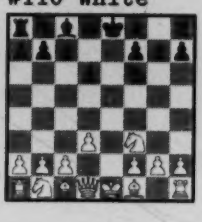
#108 White



#109 White



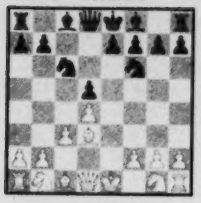
#110 White



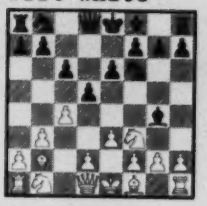
#111 White

#112 White

#113 White

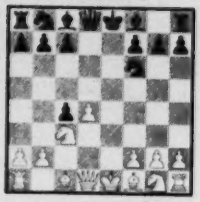


#114 White

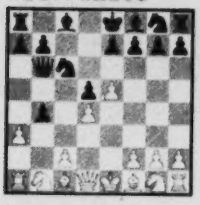


#115 White

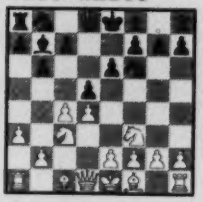
#116 White



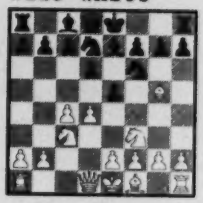
#117 White



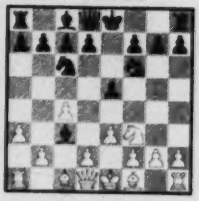
#118 White



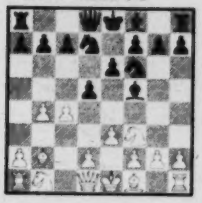
#119 White



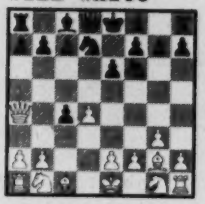
#120 White



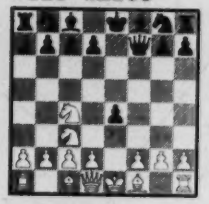
#121 White



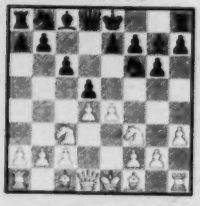
#122 White



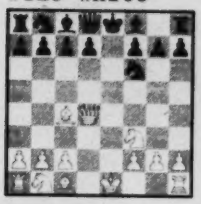
#123 White



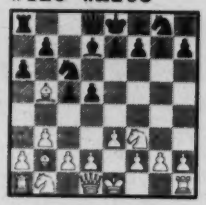
#124 White



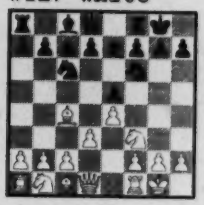
#125 White



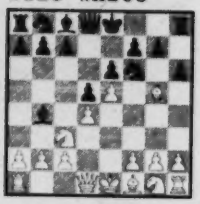
#126 White



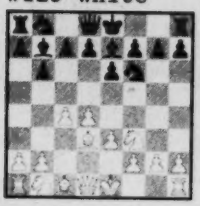
#127 White



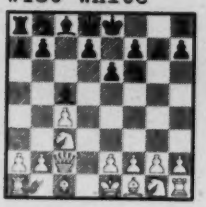
#128 White



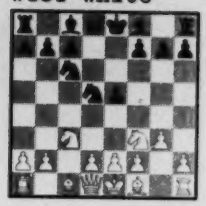
#129 White



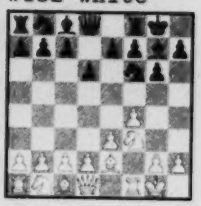
#130 White



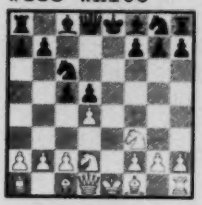
#131 White



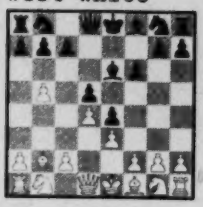
#132 White



#133 White

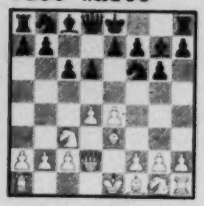


#134 White

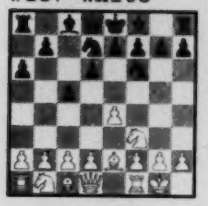


#135 White

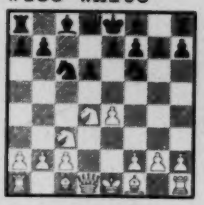
#136 White



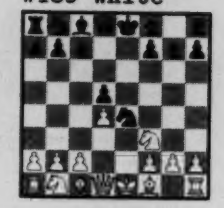
#137 White



#138 White

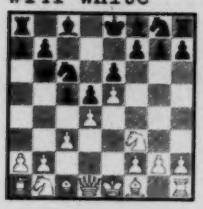


#139 White

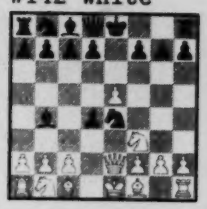


#140 White

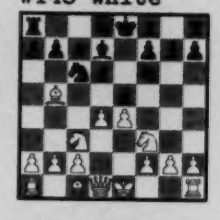
#141 White



#142 White

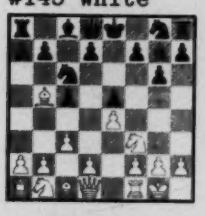


#143 White

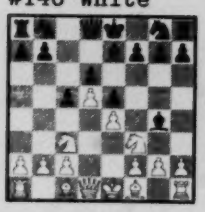


#144 White

#145 White



#146 White



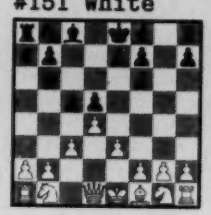
#147 White

#148 White

#149 White

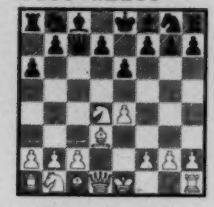
#150 White

#151 White

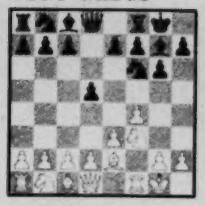


#152 White

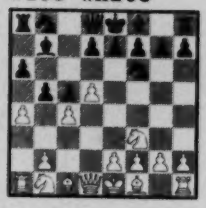
#153 White



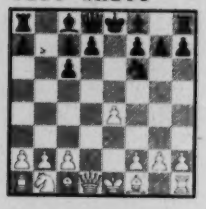
#154 White



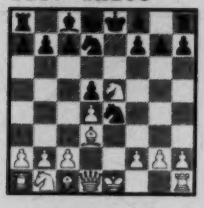
#155 White



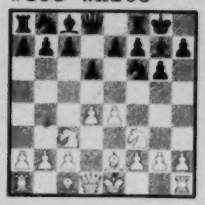
#156 White



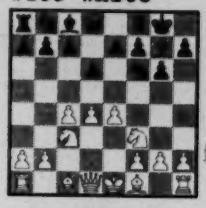
#157 White



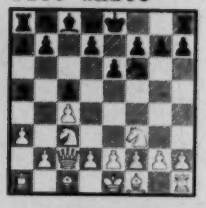
#158 White



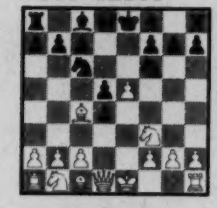
#159 White



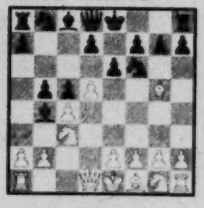
#160 White



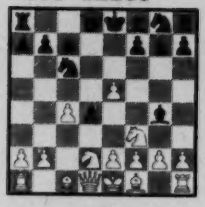
#161 White



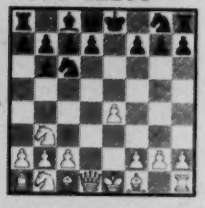
#162 White



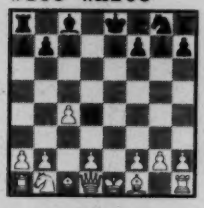
#163 White



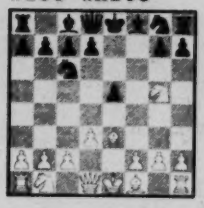
#164 White



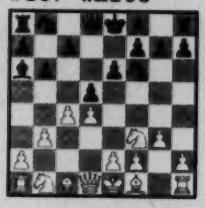
#165 White



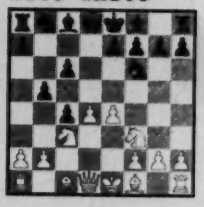
#166 White



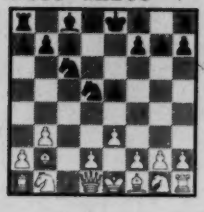
#167 White



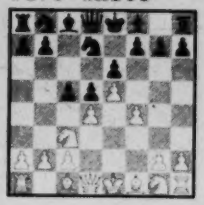
#168 White



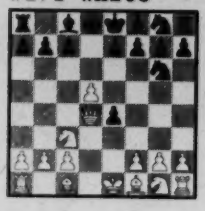
#169 White



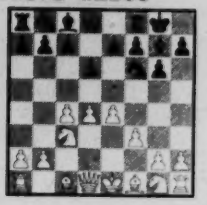
#170 White



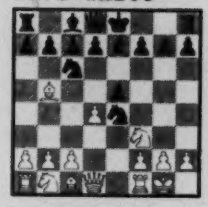
#171 White



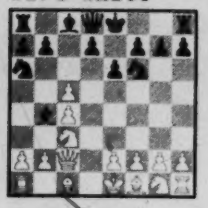
#172 White



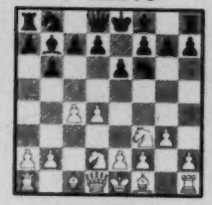
#173 White



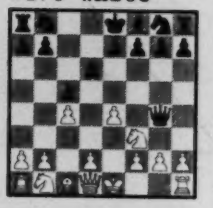
#174 White



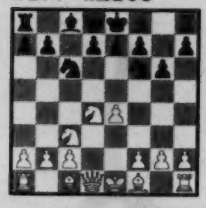
#175 White



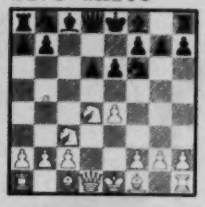
#176 White



#177 White

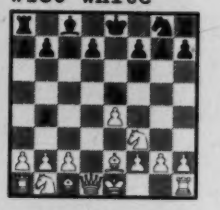


#178 White

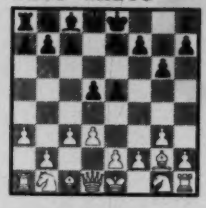


#179 White

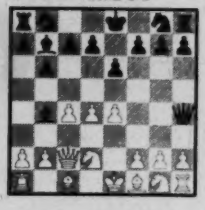
#180 White



#181 White

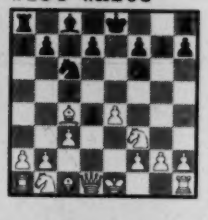


#182 White

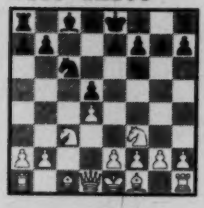


#183 White

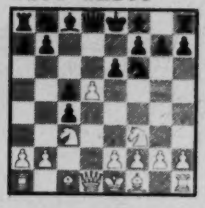
#184 White



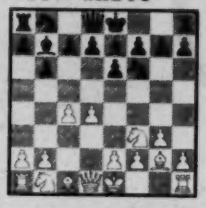
#185 White



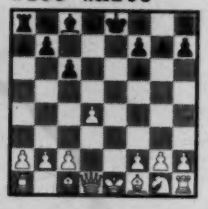
#186 White



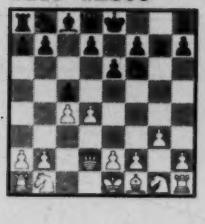
#187 White



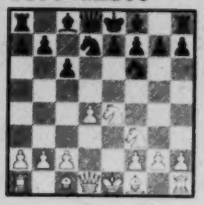
#188 White



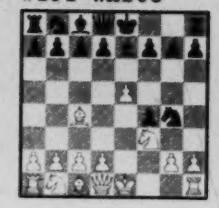
#189 White



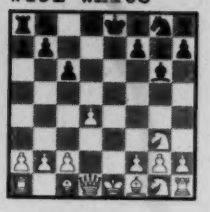
#190 White



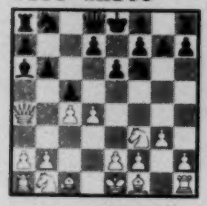
#191 White



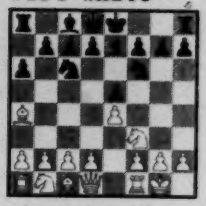
#192 White



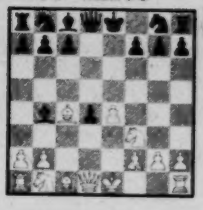
#193 White



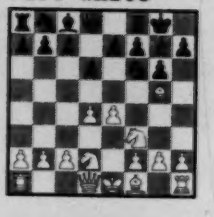
#194 White



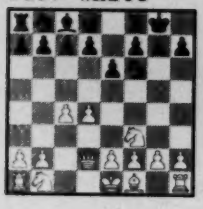
#195 White



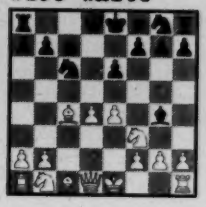
#196 White



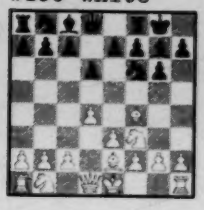
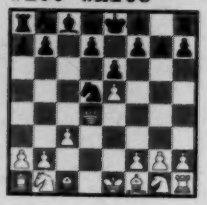
#197 White



#198 White



#199 White



#200 White